Analytische prestatieanalyse en -modellering
van superscalaire en meerdradige processors

Analytical Performance Analysis and Modeling
of Superscalar and Multi-Threaded Processors

Stijn Eyerman

Promotor: prof. dr. ir. L. Eeckhout
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. J. Van Campenhout
Faculteit Ingenieurswetenschappen
Academiejaar 2007 - 2008

UNIVERSITEIT
GENT

ISBN 978-90-8578-198-1
NUR 987
Wettelijk depot: D/2008/10.500/17

Voor ons lang verwachte wolletje,
zodat hij/zij fier mag zijn op zijn/haar papa.

Dankwoord

Dit werk is tot stand gekomen dankzij bijdragen van diverse personen en instellingen. Graag wil ik al degene die hieraan hebben meegewerkt van harte bedanken.

Als eerste wil ik in het bijzonder mijn promotor prof. Lieven Eeckhout bedanken. Het is dankzij zijn inbreng dat mijn doctoraatsonderzoek succesvol verlopen is, door het aanbrengen van nieuwe ideeën en suggesties, maar vooral door zijn onbetaalbare hulp bij het schrijven van artikels. Het is dankzij zijn ervaring dat ik erin geslaagd ben mijn onderzoeksresultaten te mogen voorstellen op belangrijke internationale fora. Ik wil hem ook bedanken om mij in contact te brengen met prof. Jim Smith.

I would also like to thank prof. Jim Smith. It is because of his many years' experience and his willingness to cooperate with Lieven and me that I was able to do the interesting research that is described in this thesis. I learned a lot from him and his contribution to my research and publications was invaluable.

Verder wil ik ook prof. Koen De Bosschere bedanken. Dankzij hem ben ik in contact gekomen met het onderzoek binnen de computerarchitectuur. Hij is ook altijd een goede hulp geweest bij het maken van presentaties en het opstellen van verslagen en aanvragen. Hij is ook samen met de andere professoren binnen de onderzoeksgroep PARIS verantwoordelijk voor het creëren van een stimulerend onderzoeksklimaat, door het voorzien in de gepaste infrastructuur en hun gewaardeerde inbreng en suggesties.

Ik wil ook al mijn bureaugenoten Veerle, Juan, Frederik, Luk, Kenneth en Filip bedanken voor de aangename werksfeer. Ik hoop dat ik met enkele van hen nog enkele jaren mag samenwerken.

Ik bedank ook alle leden van mijn doctoraatscommissie voor hun inspanning om dit werk te lezen en te beoordelen. *I thank all members of my PhD jury for their willingness to read and evaluate this thesis.*

Het Fonds voor Wetenschappelijk Onderzoek – Vlaanderen (FWO) ben ik dankbaar voor het financieren van dit onderzoek onder de vorm van een mandaat als aspirant.

De voorbije 3,5 jaar zouden natuurlijk niet zo voorspoedig verlopen zijn zonder de onmisbare steun van het thuisfront. Dank aan mijn ouders, broer, zus, schoonouders en schoonzus voor hun interesse in het verloop van mijn onderzoek, ondanks het feit dat het niet altijd gemakkelijk uit te leggen was wat ik precies deed. Speciale dank gaat uit naar mijn vrouw Evelien, die vooral de laatste weken voor het afwerken van deze thesis het soms hard te verduren had met mij, op een moment dat juist zij de meeste aandacht verdiende voor het nieuwe leven dat ze in haar draagt.

Stijn Eyerman
Gent, 13 februari 2008

Examencommissie

Prof. Daniël De Zutter, voorzitter
Decaan Faculteit Ingenieurswetenschappen
Universiteit Gent

Prof. Koen De Bosschere, secretaris
Vakgroep ELIS, Faculteit Ingenieurswetenschappen
Universiteit Gent

Prof. Lieven Eeckhout, promotor
Vakgroep ELIS, Faculteit Ingenieurswetenschappen
Universiteit Gent

Prof. Tom Dhaene
Vakgroep INTEC, Faculteit Ingenieurswetenschappen
Universiteit Gent

Prof. Geert Deconinck
Vakgroep ESAT, Faculteit Ingenieurswetenschappen
Katholieke Universiteit Leuven

Prof. James E. Smith
Department of Electrical and Computer Engineering
University of Wisconsin–Madison, USA

Prof. Erik Hagersten
Department of Information Technology
Uppsala University, Sweden

Prof. Jan Van Campenhout
Vakgroep ELIS, Faculteit Ingenieurswetenschappen
Universiteit Gent

Samenvatting

Omdat de huidige superscalaire processors zo complex zijn, is de prestatieanalyse ervan een grote uitdaging. Een processor kan meerdere instructies per klokcyclus uitvoeren, en de instructies kunnen in een andere volgorde uitgevoerd worden dan beschreven door het statische programma (out-of-order uitvoering). Daarnaast kunnen missers optreden binnen de verschillende trappen van de processorpijplijn: het ophalen van instructies kan geblokkeerd worden door missers in de instructiecache, foutieve sprongvoorspellingen veroorzaken het ophalen van instructies langs een verkeerd pad, die dan uiteindelijk geannuleerd moeten worden, en missers in de datacache kunnen de uitvoeringstijd van geheugeninstructies drastisch verlengen. Bovendien kunnen er tijdens de afhandeling van missers ook instructies uitgevoerd worden en / of andere missers afgehandeld worden. De prestatie van meerdradige processors is zelfs nog moeilijker te analyseren, omdat de programma's of draden die tegelijkertijd op de processor uitgevoerd worden dicht met elkaar verweven zijn en elkaars prestatie beïnvloeden. Daardoor is het moeilijk na te gaan welke factoren de prestatie van de individuele draden bepalen.

Het resultaat is dat processorontwerpers geen intuïtief inzicht meer hebben in hoe de prestatie van een programma dat uitgevoerd wordt op een processor tot stand komt, en wat de impact op de prestatie is van de verschillende missers die kunnen optreden binnen de processor. Daarom is men overgeschakeld op simulatie om de prestatie van een processorontwerp te evalueren. Simulatie heeft als voordeel dat het zeer flexibel is en dat de prestatiemetingen ervan zeer nauwkeurig zijn. Het is echter een zeer tijdrovend proces, het simuleren van enkele seconden van de uitvoering van een programma op een processor kan uren of dagen in beslag nemen, zelfs met de snelste simulators op de huidige snelste computers. Simulatie biedt ook een beperkt inzicht in de factoren die de totale prestatie bepalen.

Het gebrek aan inzicht bemoeilijkt ook het nauwkeurig opmeten en analyseren van de prestatie tijdens de uitvoering. Prestatietellers in hardware kunnen het voorkomen van verschillende gebeurtenissen die een invloed hebben op de prestatie opmeten, maar ze kwantificeren niet de exacte bijdrage van deze gebeurtenissen tot de totale prestatie. Ze bieden daarom weinig inzicht in de prestatie van een programma in uitvoering.

Om deze problemen te verhelpen stellen we in dit werk een analytisch prestatiemodel voor superscalaire processors voor. Het model is gebaseerd op *intervalanalyse*, dat de totale uitvoeringstijd van een programma opdeelt in geïsoleerde intervallen, afgebakend door missers. Intervalanalyse brengt de volgende inzichten aan het licht over de prestatiekost van de diverse missers:

- De kost voor een *misser in de instructiecache* is gelijk aan de tijd nodig om de instructie op te halen uit het onderliggende cacheniveau. De kost van een I-TLB misser is de tijd nodig om de paginatabel in het geheugen te consulteren.

- De kost voor een *foutieve sprongvoorspelling* bestaat uit de tijd nodig om alle instructies uit te voeren waarvan de sprong afhankelijk is, plus de tijd nodig om de front-end pijplijn van de processor opnieuw te vullen met instructies. De totale kost van een foutieve sprongvoorspelling kan dus veel groter zijn dan het aantal pijplijntrappen in de front-end van de processor, een vaak gemaakte veronderstelling.

- *Korte datacachemissers* (missers die afgehandeld worden in een tussenliggend cacheniveau en niet het hoofdgeheugen moeten raadplegen) en *instructies met een langere uitvoeringstijd* (bv. vermenigvuldigingen en delingen) veroorzaken vaak geen prestatieverlies, omdat hun latentie verborgen wordt door de out-of-order uitvoering. Enkel als ze op een lang afhankelijkheidspad liggen kunnen ze het instructievenster doen opvullen en een klein prestatieverlies veroorzaken.

- *Lange datacachemissers* (waarvan de data uit het hoofdgeheugen opgehaald moet worden) en D-TLB missers zorgen ervoor dat het instructievenster volloopt met instructies, waardoor geen nieuwe instructies meer kunnen gedispatched worden. Hoewel er door de out-of-order uitvoering nog enkele instructies kunnen uitgevoerd worden tijdens de afhandeling van de misser, kan de pres-

tatiekost van een lange datacachemisser benaderd worden door de toegangstijd naar het geheugen.

- Lange datacachemissers die dicht bij elkaar voorkomen in de dynamische instructiestroom, zodat ze samen binnen het instructievenster kunnen zitten, overlappen elkaar volledig als ze onafhankelijk van elkaar zijn. Zo verbergen ze een deel van hun kost (geheugenparallellisme). Overlappingen tussen lange datacachemissers hebben daarom een belangrijke impact op de prestatie van een processor.

Intervalanalyse geeft dus meer inzicht in hoe de prestatie van een processor bepaald wordt. Die inzichten hebben we gebruikt om een mechanistisch prestatiemodel voor superscalaire processors te ontwikkelen. Dit model kan de prestatie van de uitvoering van een programma schatten zonder gedetailleerde simulaties. Dit wordt verkregen door het aantal missers op te meten met behulp van snelle functionele simulaties, en de prestatiekost van de verschillende missers te schatten. De belangrijkste uitdagingen zijn het schatten van de kost van de foutieve sprongvoorspellingen en het bepalen van de hoeveelheid geheugenparallellisme dat uitgebuit kan worden tijdens de uitvoering. We hebben algoritmen ontwikkeld die beide karakteristieken nauwkeurig en efficiënt kunnen schatten. De gemiddelde schattingsfout is 6,9%. We demonstreren ook het nut van het mechanistisch model bij het ontwerpen van een processor door een studie van de optimale pijplijndiepte en -breedte van een processor. Een interessant resultaat van deze studie is het verband tussen optimale pijplijndiepte en de breedte van de pijplijn, namelijk dat optimale pijplijndiepte omgekeerd evenredig is met de vierkantswortel van de pijplijnbreedte.

Intervalanalyse was ook de basis voor het ontwikkelen van een prestatietellerarchitectuur die de prestatie-impact van elk van de missers opmeet. Deze informatie kan gebruikt worden om zogenaamde ‘cycle component stacks’ op te bouwen. Deze stellen de verdeling van de totale uitvoeringstijd voor in componenten die de prestatie-impact van elk type misser kwantificeren. De prestatietellerarchitectuur omvat slechts enkele honderden bits in de hardware en kan continu, en zonder de prestatie te beïnvloeden, de prestatie van een programma in uitvoering opmeten en analyseren. Vergeleken met simulatie bedraagt de maximale fout per component 4% en de gemiddelde fout is 2,5%. Het nut van nauwkeurige ‘cycle component stacks’ voor ontwerpers van compilers

demonstreren we met een studie van de prestatie-impact van verschillende compileroptimalisaties.

We hebben ook de prestatietellerarchitectuur uitgebreid naar meerdradige processors (SMT processors). Het belangrijkste probleem in meerdradige processors is de onderlinge prestatiebeïnvloeding tussen de draden, wat het moeilijk maakt om de prestatie van de individuele draden te isoleren. Daarom ontwikkelden we een mechanisme dat in staat is de cycle component stacks van elk van de individuele draden te reconstrueren alsof ze geïsoleerd op een ééndradige processor zouden uitvoeren. Daardoor kan het de voortgang van elk van de draden nauwkeurig schatten, wat nuttig is voor systeemsoftware of -hardware om in een bepaalde dienstverleningskwaliteit (bv. prestatiegarantie) te voorzien op meerdradige processors.

Summary

Due to its complexity, analyzing the performance of a modern superscalar processor is a challenging task. The processor can execute multiple instructions per cycle, and the instructions execute possibly out-of-order. In addition, various miss events can happen at different stages in the processor pipeline: the fetching of instructions can stall due to instruction cache misses, branch mispredictions cause the fetching of wrong-path instructions, which will eventually be flushed, and data cache misses can drastically delay the execution of memory instructions. Furthermore, miss event handling can be overlapped with instruction execution and/or the handling of other miss events. Multithreaded processors (e.g., simultaneous multithreading (SMT) processors) are even more difficult to analyze, since the concurrently executing threads are closely interlaced and have an impact on each other's performance.

As such, it is difficult to get an intuitive understanding of the performance of a program executing on a contemporary processor, and get insight into how big the performance impact is of the various miss events that can occur within the processor. Therefore, performance evaluation in an experimental context has shifted towards simulation. Simulation has the advantage that it is very flexible and that its overall performance estimations are very accurate. It is however very time-consuming, simulating a few seconds of real execution time can take hours or days, even with the fastest simulators running on today's fastest computers. It also provides little insight into the factors that determine overall performance.

The lack of insight also makes accurate on-line performance monitoring more difficult. Hardware performance counters can measure various events that have an impact on performance, but they do not quantify the contribution of these events to overall performance. There-

fore, the insight they provide into the performance of a program executing on a processor is limited.

To overcome these problems, we propose an analytical performance model for superscalar processors. The model is based on *interval analysis*, which divides total execution time into isolated intervals, delimited by miss events. Interval behavior is observed most clearly in the dispatch behavior of a program executing on a processor. Dispatch has an on/off behavior, which makes it easier to delimit intervals, and to calculate the penalty of a miss event. Interval analysis reveals the following insights regarding the performance penalty of the various miss events:

- The *instruction cache miss* penalty equals the time needed to fetch the instruction from the appropriate level in the memory hierarchy. The penalty of an I-TLB miss equals the time needed to consult the page table in memory.

- The *branch misprediction* penalty consists of the branch resolution time, i.e., the time needed to execute the dependency path leading to the mispredicted branch, plus the front-end pipeline refill time. Due to the branch resolution time, the total branch misprediction penalty can be much larger than the front-end pipeline depth, which is often assumed to be the penalty of a branch misprediction.

- *Short data cache misses* and *medium-latency instructions* typically do not incur a performance penalty, because their latencies are hidden by out-of-order execution. Only if they are on long dependency paths, they can cause the instruction window to fill up and incur a small penalty.

- A *long-latency load miss* (L2 D-cache miss or D-TLB miss) causes the instruction window to fill up completely and therefore stalls the dispatch of new instructions. Although the out-of-order execution allows some instructions to execute while the load miss is pending, the penalty of a long-latency load miss can be approximated by the memory access time.

- The penalties of long-latency load misses that can concurrently reside within the instruction window and that are independent of each other, completely overlap (memory-level parallelism or MLP). Overlapping load misses therefore have a substantial impact on processor performance.

Using the insights provided by interval analysis, we develop a mechanistic performance model that is able to estimate the performance of a program executing on a superscalar processor without the need for detailed simulations. It does so by counting the number of miss events, using fast functional simulation, and estimating the penalties of the various miss events. The main challenges are the estimation of the branch resolution time to calculate the branch misprediction penalty, and the estimation of the MLP that is exploited during the execution. We develop algorithms to estimate both characteristics in an efficient and accurate way. The average overall performance estimation error is 6.9% for a baseline four-wide superscalar out-of-order processor. We illustrate the use of the mechanistic model in processor design through a study of optimal processor pipeline depth and width. This results in an interesting relationship between optimal depth and width, namely optimal pipeline depth is inversely proportional to the square root of pipeline width.

Interval analysis also forms the basis for the development of a performance counter architecture that measures the performance impact of each of the miss events. This information can be used to construct cycle component stacks, i.e., the decomposition of total execution time into various cycle components that quantify the performance impact of each type of miss event. The performance counter architecture incurs only a few hundred bits of extra storage and continuously and uninvasively monitors the performance of a running program. The maximum cycle component error compared to a simulation-based approach is 4% and the average error is 2.5%. The value of accurate cycle component stacks for compiler builders is shown through a study of the performance impact of compiler optimizations.

We also extend the performance counter architecture to SMT processors. The main problem in multi-threaded processors is the mutual performance impact between the threads, which makes it difficult to isolate the performance of the individual threads. We therefore develop a mechanism that constructs the cycle component stacks of each of the threads during SMT execution, as if they were executing on a single-threaded processor. It can estimate the single-threaded progress of the individual threads, which is useful for system software or hardware to provide quality of service on multi-threaded processors.

Contents

List of Tables

List of Figures

List of Abbreviations

ANTT	Average Normalized Turnaround Time
BMT	Back-end Miss Event Table
BTB	Branch Target Buffer
CIC	Committed Instructions Counter
CDIC	Committed Dependent Instructions Counter
CISC	Complex Instruction Set Computer
CMOS	Complementary Metal Oxide Semiconductor
CMP	Chip Multiprocessor
CPI	Cycles Per Instruction
CPU	Central Processing Unit
D-Cache	Data Cache
DLP	Dependent Load Pointer
DL1	Level 1 Data Cache
DL2	Level 2 Data Cache
D-TLB	Data Translation Lookaside Buffer
ECL	End of Cacheline
FMT	Front-end Miss Event Table
I-Cache	Instruction Cache
ICOUNT	Instruction Count (fetch policy)
IFSC	In-Flight Slots Counter
IFWS	In-Flight Waiting Slots
ILP	Instruction Level Parallelism
IL1	Level 1 Instruction Cache
IL2	Level 2 Instruction Cache
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
I-TLB	Instruction Translation Lookaside Buffer
LSQ	Load Store Queue
L1	Level 1 (cache)
L2	Level 2 (cache)
L3	Level 3 (cache)

MIPS	Million Instructions Per Second
MLP	Memory-Level Parallelism
ORB	Output Register Bitmap
QoS	Quality of Service
RAW	Read After Write (dependency)
RISC	Reduced Instruction Set Computer
ROB	Reorder Buffer
sFMT	shared Front-end Miss Event Table
SMT	Simultaneous Multithreading
SOE	Switch On Event
STP	System Throughput
TLB	Translation Lookaside Buffer
TPI	Time Per Instruction
TPR	Thread Progress Register
V-ROB	Virtual Reorder Buffer
WAR	Write After Read (dependency)
WAW	Write After Write (dependency)
WB	Write Buffer (stall)

Chapter 1

Introduction

The purpose of computing is
insight, not numbers.
Richard Hamming

1.1 Motivation

Processor architecture has undergone substantial enhancements over the few decades of its history. Going from the early processors in the 1950s, that ran at a few kilohertz, to today's 4.7 gigahertz[1] high-performance computers, computer architects have come up with many ideas to improve both the speed and the efficiency of the processor. The driving force behind this evolution is the spectacular progress in CMOS technology, allowing more and more, and increasingly faster transistors being integrated on a single chip, as predicted by Moore's law. It is the task of a computer architect to translate these technological advances into better performing computers.

Several major microarchitecture enhancements have become state-of-the-art in commercial processor design. One of them is pipelining, allowing multiple instructions in the processor, each executing a different phase of their computation. This reduces the complexity of each phase compared to sequential non-pipelined processors, allowing shorter cycle times and thus higher clock frequencies. Other enhancements that boosted the efficiency of the processor are superscalar architectures that execute multiple instructions per cycle, and

[1] IBM POWER6 [61]

out-of-order issue, that starts the execution of an instruction as soon as its input data is available, instead of using the ordering enforced by the programmer or compiler. Effective branch predictors were introduced to speculatively execute instructions past not yet resolved indirect and conditional branches. Multiple levels of cache memory and the prefetching of shortly needed data try to overcome the memory gap.

While these enhancements meant a huge progress in processor performance, they definitely did not simplify its design and analysis. Insight tends to be inversely proportional to complexity, and the details of each individual part of the processor can overshadow the ability to have an integrated view of the processor, which makes it difficult to analyze the impact each part has on overall performance. In addition, to complicate matters even further, performance is very much application-dependent. For example, the accuracy of a branch predictor can be easily assessed by analyzing the outcome of all branches of an application during its execution, but the performance impact of a branch misprediction depends on both the sizing of other processor parts and the application being executed. Likewise, an application can be developed such that it has many independent instructions to efficiently exploit the superscalarity of the processor, but if its memory access behavior is problematic, there will be many cache misses and the processor will not be able to exploit the program's high instruction-level parallelism (ILP).

The recent shift towards multi-threaded processors (e.g, simultaneous multithreading (SMT)) has even worsened the problem. By executing two or more threads simultaneously on one processor, performance analysis gets even more complicated because of the complex interactions between the threads. In an SMT processor, all threads compete for the same resources, which has its impact on performance. For example, if one thread occupies most of the available instruction window entries, the other threads can have less in-flight instructions, which degrades their performance.

The complex interactions between processor architecture and application characteristics, such as miss events (branch mispredictions, cache misses, etc.) and ILP, make it difficult for computer architects to build intuitive performance models, which are needed to gain insight, and to estimate and evaluate the performance impact of innovations. This has a substantial impact on the processor design process, where a designer tries to find the best performing processor within a given set

of design constraints, such as a maximum chip area and a maximum power budget. The lack of insight also complicates performance analysis at runtime, where users and system software (operating systems and virtual machines) need to have an accurate and detailed view on which aspects determine the performance of the running application(s). Contemporary approaches for analyzing performance at design time and at run time are described in the next two sections, along with their limitations.

1.1.1 Performance analysis at design time

Due to the lack of an intuitive but accurate performance model, performance evaluation in computer architecture research and development has shifted towards simulation. A software simulator models the processor's behavior, and performance is evaluated through the simulation of some representative benchmarks.

Simulation has the main advantage that its performance results are relatively accurate, depending on the degree of abstraction of the simulator. It is also very flexible (a parameterized simulator can simulate a large range of processor architectures), and its development is much cheaper than building hardware prototypes. However, it has two main disadvantages: it is becoming too time-consuming and provides limited insight. Since a simulator has to execute thousands of instructions in order to simulate one clock cycle of processor execution, it is much slower than the execution on the real hardware: simulating a few seconds of real execution time can take hours or days, even on today's fastest simulators running on today's fastest machines. The more complexity is added to the processor, the slower is the simulator. The more instructions the benchmark executes dynamically, the longer it takes to simulate the benchmark. The more benchmarks that are needed to characterize the typical workload of the processor, the more simulations need to be done. The larger the design space, the more design points need to be evaluated to find a balanced processor architecture configuration. Using simulation for designing processors becomes infeasible because it is too time-consuming.

Moreover, although a simulator can produce some interesting numbers, such as miss rates, resource utilization numbers as well as overall performance estimates, it provides little or no insight into what the impact is of the various processor architecture structures on performance

and their mutual interaction. In addition, answering what-if questions using a simulator is relatively hard, e.g., is a branch misprediction rate of 10% more harmful than a 15% level-1 data cache miss rate, or vice versa?

1.1.2 Performance analysis at runtime

The lack of insight also has an impact on on-line performance monitoring tools. Such tools detect and count various events while an application is executing on a processor, and present them to the user in order to analyze the performance. This is useful for processor designers to validate early stage design options on a hardware prototype. It also provides interesting information for software and compiler builders for analyzing the performance of their applications on an existing processor. Detailed performance information is also needed by system software for making the correct decisions in order to meet preassigned performance requirements or to optimize overall performance.

An intuitive way of analyzing and visualizing the performance of a computer program running on a microprocessor, which goes back to the early days of computing, are cycle component stacks (also called “CPI stacks”, or breaking up overall CPI into “CPI adders”). These stacks divide total execution time into distinct components, each reflecting the number of cycles spent in executing instructions, handling cache misses, resolving branch mispredictions, etc. Current hardware performance counters can count the number of dynamically executed instructions, the number of miss events, and the number of resource stalls, but they do not quantify the impact these numbers have on overall performance. This means that they are unable to construct accurate cycle component stacks. They also do not take into account that miss events can overlap on an out-of-order processor, which further complicates accurate and unambiguous cycle accounting.

Performance analysis on multi-threaded processors is even more challenging. Since concurrently executing threads are closely interlaced, it is difficult to separate the performance impact of miss events on each of the individual threads and the impact they have on each other. The lack of an intuitive performance model makes it impossible to build performance counters that can perform accurate performance analysis.

1.2 Thesis statement

An analytical performance model helps to better understand the performance of current superscalar out-of-order and multi-threaded processors. In fact, in this dissertation, **we show how an intuitive and accurate mechanistic performance model can be built based on first principles**, i.e., by looking into the mechanisms in the processor that have an impact on performance. This mechanistic model enables accurate performance analysis at design time as well as at runtime.

First, it provides more insight into what factors have an impact on the overall performance of a program executing on a processor, and quantifies the impact of these factors. These insights are useful for processor designers and users, to be able to reason about the possible performance impact of new hardware innovations or program code optimizations.

Second, an analytical model also enables faster performance estimations compared to detailed simulations, because it can determine performance using processor and program characteristics, without the need to track the detailed behavior of the individual instructions. This enables the processor designer to do fast design space exploration in order to find the design that performs the best within its design constraints.

Third, a performance model can indicate how to construct meaningful cycle component stacks. The model can be used as a guidance to construct a hardware performance counter architecture that enables the measurement of accurate cycle component stacks. By constructing a performance counter architecture in a top-down manner, i.e., based on a model instead of an ad hoc implementation, the actual performance impact of different events within the microprocessor can be measured more accurately.

1.3 Contributions in this dissertation

This work presents a new analytical performance model for superscalar out-of-order and multi-threaded processors. This model can serve as a basis for performance analysis and modeling tools. More specifically, the contributions of this work are the following.

Interval analysis

We propose an analytical performance model called *interval analysis*, that is accurate as well as easy to understand. It is based on the observation that performance can be analyzed by considering the dispatch behavior of a program executing on a processor. In the absence of miss events, dispatch rate is (almost) always equal to the designed dispatch width, and a miss event causes dispatch rate to fall to zero abruptly. This provides a sharp delimiter for defining busy time (the processor is doing useful work) and miss event penalties (the performance cost of miss events). By doing so, interval analysis provides insight into the performance impact of the various types of miss events.

A discussion of interval analysis applied to the better understanding of the penalty of branch mispredictions has been published in:

- Stijn Eyerman, James E. Smith and Lieven Eeckhout. "Characterizing the Branch Misprediction Penalty." In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE Computer Society. 2006. pp 48-58.*

A more complete elaboration of interval analysis is published as a part of the papers describing the other contributions.

A mechanistic performance model

Based on the insights provided by interval analysis, we develop a mechanistic performance model. This is done by counting the miss events using specialized cache and branch predictor simulators, and estimating the penalty of each miss event. The penalties are estimated based on the insights provided by interval analysis, using parameters that describe the structure and sizing of the processor, and an analysis of the program's dependency and locality characteristics. It differs from prior work in that it does not need detailed simulations to determine empirical parameters or steady-state performance. It also provides a new and more accurate way for calculating the branch misprediction penalty, and it shows how interval behavior has an impact on the dispatch inefficiency and the branch misprediction penalty. The average estimation error is 6.9% for a baseline four-wide superscalar out-of-order processor.

The model consists of equations that can be used for optimizing designs without needing to evaluate every possible design point through

simulation. This is illustrated through an optimal processor pipeline depth versus width study, in which both pipelined and non-pipelined cache and memory accesses are modeled. The impact of deepening and widening the processor pipeline on each of the performance components is quantified and a relationship between optimal pipeline depth and width is deduced, namely optimal pipeline depth is inversely proportional to the square root of pipeline width.

An extensive discussion of this performance estimation method is described in:

- Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis and James E. Smith. "Mechanistic Performance Modeling for Studying Resource Scaling in Superscalar Processors." Submitted to *ACM Transactions on Computer Systems*.

A hardware performance counter architecture for accurate cycle accounting

Interval analysis can also be used as a basis for accurate performance analysis. It divides total execution time into a base component where useful work is done and different miss event penalties. This information is very useful for hardware, software and compiler builders, since it provides an intuitive representation of a program's execution, revealing possible performance bottlenecks and optimization opportunities. We have constructed a new hardware performance counter architecture for superscalar out-of-order processors that is able to continuously and uninvasively monitor performance, providing an accurate execution time division into its cycle components at any moment. The maximum cycle component error compared to a simulation-based approach is 4% and the average error is 2.5%.

This work has been published in:

- Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis and James E. Smith. "A Performance Counter Architecture for Computing Accurate CPI Components." In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM. 2006. pp 175-184.*

- Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis and James E. Smith. "A Top-Down Approach to Architecting CPI Component

Performance Counters." In *IEEE Micro, Special Issue on Top Picks of the 2006 Microarchitecture Conferences. IEEE Computer Society. Vol. 27. 2007. pp 84-93.*

A case study demonstrating the use of this performance counter architecture for studying the performance impact of compiler optimizations has been published in:

- Stijn Eyerman, Lieven Eeckhout and James E. Smith. "Studying Compiler Optimizations on Superscalar Processors through Interval Analysis." In *Proceedings of the 2008 International Conference on High Performance Embedded Architectures & Compilers (HiPEAC). 2008. pp 114-129.*

Cycle accounting in simultaneous multithreading processors

Simultaneous multithreading (SMT) processors are built to execute multiple programs (or threads) simultaneously on a single processor core. Because the concurrently executing threads share many of the processor's resources, it is difficult to isolate the performance of the individual threads and quantify the interaction between the co-executing threads. We have extended the performance counter architecture for single-threaded processors to SMT processors, providing an accurate performance analysis for each of the individual threads. This new performance counter architecture quantifies the performance impact of miss events as well as the performance impact due to interactions between co-executing threads. It is able to monitor the progress of each of the individual threads during multi-threaded execution, in terms of the number of cycles they would have executed on a single-threaded processor. This cycle accounting architecture can be used to establish quality of service on an SMT processor, by providing per-thread progress information to system software or by guiding a hardware implemented policy to enforce a certain performance target to each of the individual threads, irrespective of the co-executing threads.

1.4 Other research activities

Besides the contributions mentioned above, we also performed research on the following topics—these research results are not described

in this dissertation, but we refer the interested reader to the respective papers and to the appendices in this dissertation.

Processor design space exploration

We have explored and evaluated several single-objective and multi-objective search algorithms in the context of processor design space exploration. We show how processor optimization can be done efficiently by using a fast simulation technique (in casu statistical simulation) in combination with a heuristic search algorithm. The optimum found can then be further refined using a search algorithm needing less evaluation steps through detailed simulation.

This study is published in:

- Stijn Eyerman, Lieven Eeckhout and Koen De Bosschere. "Efficient Design Space Exploration of High Performance Embedded Out-of-Order Processors." In *Proceedings of Design, Automation and Test in Europe (DATE). EDAA. 2006. pp 351-356.*

A memory-level parallelism aware fetch policy for SMT processors

A fetch policy for an SMT processor decides each cycle from which thread instructions should be fetched. It should be aware of the resource usage of each of the threads to make an intelligent decision. An important parameter is the occurrence of long-latency load misses, because a long-latency load miss can cause the instruction window to fill up with instructions of a single thread, thereby starving the other threads. Prior work has overcome this by stalling threads experiencing long-latency load misses and flushing instructions to free instruction window entries, but the considerable performance impact of multiple long-latency loads occurring in parallel (memory-level parallelism or MLP) was neglected. We designed an MLP-aware fetch policy that prevents threads from losing their available MLP, i.e., instructions are only flushed if they do not reveal any MLP.

This work has been published in:

- Stijn Eyerman and Lieven Eeckhout. "A Memory-Level Parallelism Aware Fetch Policy for SMT Processors." In *Proceedings of the Thirteenth International Symposium on High-Performance Com-*

puter Architecture (HPCA). IEEE Computer Society. 2007. pp 240-249.

- Stijn Eyerman and Lieven Eeckhout. "Memory-Level Parallelism Aware Fetch Policies for Simultaneous Multithreading Processors." Accepted under revision in *ACM Transactions on Architecture and Code Optimization*.

Performance metrics for SMT processors

Meaningful performance metrics are crucial for evaluating and comparing different design options. For single-program performance, execution time is the metric of choice. For multi-program workloads, e.g., different programs executing on a multi-threaded processor, there is no consensus on what metrics should be used. We advocate that performance metrics for architects should be developed from system-level objectives in a top-down fashion. We propose two multi-program performance metrics, a system-oriented one (system throughput) and a user-oriented metric (average normalized turnaround time). They are both closely related to previously proposed metrics, but we provide a system-level meaning to them.

This work is described in:

- Stijn Eyerman and Lieven Eeckhout. "System-Level Performance Metrics for Multi-Program Workloads." Accepted for *IEEE Micro, issue May/June 2008.*

1.5 Thesis outline

This dissertation is outlined as follows. The next chapter will give an overview of the microarchitecture of a superscalar out-of-order processor and a simultaneous multithreading processor, as well as an introduction to performance estimation and simulation. In Chapter 3, interval analysis is introduced and explained in detail. Chapter 4 shows how to construct a mechanistic superscalar processor performance model based on interval analysis. Chapter 5 uses these insights to construct a new performance counter architecture that continuously monitors and analyzes performance, by accurately measuring the performance impact per miss event type. This architecture is subsequently extended for simultaneous multithreading processors in Chapter 6,

and it is shown how it can be used to enforce quality of service on these processors. Finally, Chapter 7 summarizes the conclusions and gives some suggestions for future work. Appendices B and C summarize the work about performance metrics for multi-program workloads running on multi-threaded hardware and the MLP-aware fetch policy for SMT processors, respectively.

Chapter 2

Background

Computer architecture?
Is that about designing the box around a computer?
A person in my village,
asking about my research topic.

To provide some background information, this chapter revisits the processor architecture of contemporary microprocessors as well as an introduction to performance evaluation.

2.1 Processor architecture

In this section, a survey is given of the processor architectures that are studied in this thesis: superscalar out-of-order processors and simultaneous multithreading processors.

2.1.1 Superscalar out-of-order processors

Current general-purpose high-performance processors are mostly based on the paradigm of a superscalar out-of-order scheduling architecture [84]. This architecture is designed to extract instruction-level parallelism (ILP), by enabling each pipeline stage to handle multiple instructions per cycle, and by selecting independent instructions out of a large window of instructions, irrespective of program order. By doing so, the number of instructions executed per cycle (IPC) is pushed up, and together with excessive pipelining to boost clock frequency, it is (one

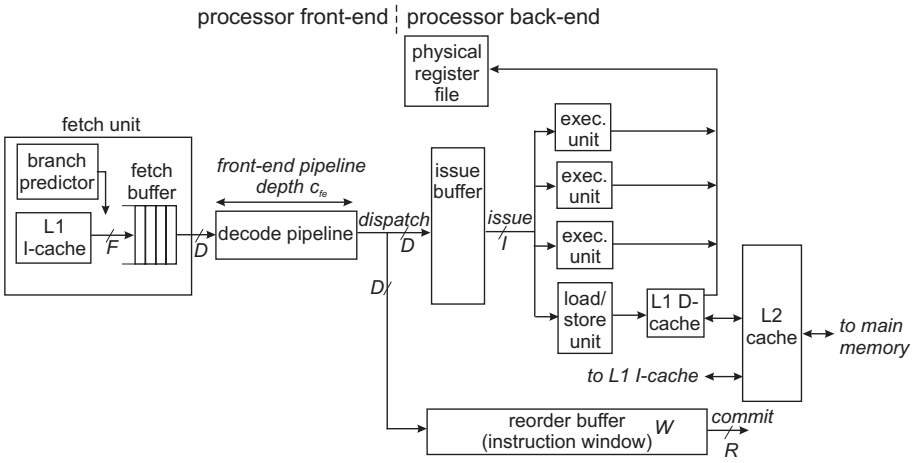

Figure 2.1: A generic superscalar out-of-order processor.

Table 2.1: Processor parameter definitions, notations and baseline values.

Parameter	Notation	Baseline value
Fetch width	F	8
Decode / dispatch width	D	4
Front-end pipeline depth (cycles)	c_{fe}	5
Issue width	I	4
Commit / retire width	R	4
Instruction window / reorder buffer size	W	128

of) the best performing processor architectures for general-purpose use. Although simpler single-issue and / or in-order processors have been used in application-specific systems as well as embedded systems to save chip area and power consumption, more and more embedded applications demand higher performance as well, which can only be provided by more complex processor architectures.

A generic superscalar out-of-order processor is illustrated in Figure 2.1. Table 2.1 introduces some definitions and notations about the sizing of processor structures that are used throughout the remainder of this thesis. It also indicates the baseline values for these parameters that reflect the processor configuration used in most of the experiments reported in this thesis (for a more detailed report on the baseline processor configuration, see Appendix A).

The processor consists of two main parts: the front-end, which supplies new instructions to the instruction window, and the back-end, which executes instructions selected from the instruction window, and which updates the architectural state of the processor. These parts are discussed in the next two sections.

Processor front-end

The processor front-end has two main tasks: fetching instructions from memory, and transforming (decoding) the packed instructions into a more convenient format for further processing through the processor pipeline. The *fetch unit* fetches instructions from the level-1 instruction cache (L1 I-cache). The L1 I-cache is a cache of limited size that contains recently fetched instructions. If an instruction is not available in the instruction cache, an instruction cache miss occurs, and fetch is suspended until the instruction is fetched from a lower level cache (level-2 (L2) or possibly level-3 (L3) cache) or main memory.

The number of instructions fetched per cycle, F, should be larger than or equal to the width of the rest of the processor front-end pipeline. An indication to make the fetch width larger than the rest of the front-end pipeline is the fact that due to branches and a suboptimal instruction layout, a cacheline may not always contain enough useful instructions to support the processor's front-end pipeline width. If the next instruction to fetch is on another cacheline, instruction fetch is suspended for this cycle, the cacheline is fetched from the instruction cache in the next cycle, and instruction fetch can resume. To offset cycles where fewer instructions than the width of the front-end pipeline can be fetched, the fetch unit is designed to be wider, with a fetch buffer to moderate the flow of instructions from the instruction cache into the front-end pipeline.

The outcome of conditional branches and the target of indirect jumps is unknown in the processor front-end—the outcome is not known until their execution, which happens in the back-end. Waiting for that would be too inefficient, so branch directions are predicted using a branch predictor, and their targets using a branch target buffer (BTB). Fetch continues at the predicted target. As with any predictor, some predictions may be wrong. When a branch is executed and a misprediction is discovered, the processor flushes all instructions along the mispredicted path, and restarts fetching at the next correct-path

instruction. The associated performance loss, i.e., the branch misprediction penalty, will be discussed in the next chapter.

The *decode pipeline* extracts valuable information from the packed instructions, e.g., the instruction's type (e.g., addition, multiplication, memory read / write, etc.), and its input and output registers. If the implemented instruction set is of the CISC-type (complex instruction set computer, e.g., the x86 instruction set), it also breaks up the complex instructions into RISC-like (reduced instruction set computer) micro-operations.

The decode pipeline also renames the input and output registers of the instructions, such that all artificial dependencies—write-after-read (WAR) and write-after-write (WAW)—are eliminated, while retaining the real dependencies—read-after-write (RAW). Rename registers are chosen from a physical register file, which contains at least as many (but usually more) registers as there are architectural registers.

An important parameter of the front-end pipeline is its depth, c_{fe}, i.e., the number of cycles an instruction needs to traverse the front-end pipeline. As will be shown in the next chapters, the front-end pipeline depth has a considerable impact on the performance of a processor. The entering of instructions from the front-end pipeline into the instruction window is called *dispatch*. We will refer to D as the decode / dispatch width throughout the thesis.

Processor back-end

At the heart of the out-of-order processor is the *instruction window*, containing all in-flight instructions. The size of the instruction window is one of the most important parameters of the processor: the more instructions in the instruction window, the more ILP can be extracted and the better overall performance. The instructions in the instruction window can be divided in two parts: instructions that have been executed, and instructions waiting for input data to be executed. The instruction window has to keep track of the status of each instruction, and each cycle, a number of instructions that are ready to execute (i.e., whose dependencies are resolved) is being selected from the instruction window. This enables out-of-order execution in a data-flow manner, i.e., instructions are executed when their input operands are available. Ready instructions are then *issue*d to one of the multiple execution units, at most I (issue width) instructions per cycle. For efficiency reasons, the in-

struction window is often implemented using two separate structures: the *reorder buffer* (ROB) holding the status of all in-flight instructions and preserving program order to ensure precise interrupts, and the *issue buffer*, containing only the not yet issued instructions.

Execution units perform the actual execution of the instruction, e.g., an addition, logical operation, multiplication, etc. Special execution units are the load and store units. They look after the communication with the data cache: loads read data, and stores write new data. Special attention has to be paid to the correct ordering of loads and stores when they (could) access the same memory location: a load to the same address as a previous store should not be executed before that store. Several optimizations can be implemented here, e.g., forwarding data from a pending store to a load at the same memory location without accessing the data cache, and speculatively executing loads before prior stores (load bypassing). As is the case for the instruction cache, the level-1 data cache (L1 D-cache) also contains a limited set of all program data, i.e., the most recently addressed data and its neighboring data. Data cache misses can substantially lengthen the execution latency of load instructions, since they have to wait until the data is returned from the lower level cache or memory.

The last pipeline stage is *commit*. Here, instructions leave the processor and update the architectural state (architectural registers and data stores in memory). This has to be done in program order to ensure precise interrupts. The oldest in-flight instruction—the one at the head of the ROB—has to be committed first. If it is not yet executed, the commit stage stalls. Unlike the issue stage, which starts the execution of the instructions (possibly) out-of-order, the commit stage updates the architectural state in-order.

The out-of-order scheduling paradigm is very well adopted in commercial processors. Examples range from the Intel Pentium Pro (1995) [77] up to the current Intel Core 2 [39]; from the IBM POWER1 (1990) [40] up to the IBM POWER5 [83] (current POWER6 has re-shifted to in-order execution to obtain higher clock frequencies while limiting power consumption [61]); from the AMD K5 (1996) [15] up to the current AMD K10.

2.1.2 Simultaneous multithreading processors

There are two main reasons why a single-threaded processor is often inefficiently used. Miss events (instruction and data cache misses, and branch mispredictions) interrupt the supply of new instructions or delay the execution of instructions, and in some cases applications display not enough instruction-level parallelism to support the whole width of the processor. Therefore a processor can be extended to support the simultaneous execution of multiple applications or threads [98]. Instructions of two different threads are per definition independent (except for synchronisation and/or enforced ordering of memory accesses in multi-threaded applications), and when one thread has a miss event, the other threads usually can still fetch and execute instructions, keeping the processor busy.

In order to support simultaneous multithreading (SMT) on a superscalar out-of-order processor, some extensions have to be implemented. First, architectural state has to be stored for every thread. So we need multiple program counters as well as storage for all architectural registers per thread. In practice, this usually means that the physical register file has to be enlarged. Other structures can also be duplicated for performance reasons, in particular those containing thread specific information, such as the return address stack or the branch history register. Second, each processor resource can now process instructions from different threads, so not only are thread identifiers needed per instruction, but each processor part should now make decisions on what thread should be processed first. It should distribute the available bandwidth (e.g., fetch, dispatch, execution and commit bandwith) among the threads. The most obvious example is the fetch unit, which has to decide each cycle from which thread(s) to fetch instructions. Appendix C discusses fetch policies for SMT processors.

Another design option is to decide how storage resources are shared. The instruction window for example can be partitioned, such that each thread can only use a part of the entire window size. This can however degrade efficiency: a thread needing more entries cannot use the empty entries of another thread. Dynamically sharing these resources can improve efficiency, but has the risk of one thread clogging all resources, thereby starving other threads. Sharing predictors and caches can also degrade performance. A cache line from one thread can be evicted by a cache line of another thread, introducing extra miss events for the first thread. Branch predictor entries can also be

shared by branches of different threads, introducing additional branch mispredictions due to contention.

Examples of commercial superscalar SMT processors are the Alpha EV8 [19], the Intel Pentium 4 with Hyper-Threading Technology [5] and the IBM POWER5 [83].

2.2 Performance evaluation techniques

Having discussed what contemporary processors look like, we now discuss how these processors are being designed at the architectural level. This will enable a better understanding of the true value of the analytical model proposed in this thesis.

Accurate performance evaluation is a crucial part of computer architecture research and design. It is only by comparing performance results that the effectiveness and efficiency of an innovation can be assessed. Inaccurate performance estimations can lead to wrong conclusions by possibly favoring poor design options.

A widely used approach to obtain performance results is simulation, i.e., using software that models the behavior of a processor under design. The accuracy of a simulator is determined by its level of detail. A detailed simulator produces very accurate performance results, up to being cycle-accurate, but is very slow. To simulate one cycle of hardware execution, a software simulator (which is essentially a sequential process, as opposed to hardware that can perform many operations in parallel) requires the execution of thousands of instructions, which slows it down extensively compared to real hardware execution. Therefore, several simulation techniques have been developed, see Table 2.2. Each simulation technique represents a different trade-off between development time, accuracy and evaluation time.

Functional simulation only models the functional characteristics of an instruction set. It simulates the instructions one by one, taking the input operands and calculating the output values, but it does not model the exact timing of the instructions entering and leaving the processor. Therefore its accuracy in terms of performance estimation is poor, but due to its low level of detail, it is very fast (e.g., 7 MIPS (million instructions per second) for sim-profile, a functional simulator of the widely used SimpleScalar toolset [4]). Since only one functional simulator has to be developed per instruction set architecture (ISA), it has a very long

Table 2.2: Comparing simulation and modeling techniques.

Technique	Development time	Accuracy	Evaluation time
Functional simulation	Excellent	Poor	Good
Specialized cache and predictor simulation	Good	Poor	Good
Detailed simulation	Poor	Excellent	Poor
Sampled simulation	Poor	Good to excellent	Fair
Statistical simulation	Good	Good	Very good
Analytical modeling	Excellent	Good	Excellent

lifetime, which reduces its development cost. Functional simulation is useful to generate instruction and address traces that can be used by other simulation techniques. It can also be used to analyze the dynamic characteristics of a program, e.g., instruction mix and inter-instruction dependency characteristics, as we will do for computing some of the input parameters to the performance model presented in Chapter 4.

Specialized cache and predictor simulation combines a functional simulator with a cache or branch predictor simulator. The functional simulator extracts instruction / data addresses or branch outcomes, and feeds them into the specialized simulator. Performance is typically evaluated in terms of miss rates, i.e., how many instructions miss in the instruction cache, how many memory instructions miss in the data cache or how many branches are mispredicted. While these miss rates are estimated quite well (although miss rates can also be dependent on exact timing information and speculation), cache and branch predictor behavior is only a part of overall processor performance. Hence accuracy is poor, but since they simulate small parts of the processor in isolation, their evaluation time is good (4 to 5 MIPS for specialized simulators of the SimpleScalar toolset [4]), and the development time is reasonable.

Detailed simulation models the whole microarchitecture of the processor in detail. It is also often referred to as cycle-by-cycle simulation, i.e., each cycle the operations performed by the processor are modeled in detail. This is the most accurate way to obtain performance results (from an architectural point of view, some abstractions are however still made with respect to the circuit-level details). It can be trace-driven, which means that the input is an instruction trace generated by a functional simulator, or execution-driven, where functional simulation is combined with timing simulation. The advantage of trace-driven simulation is that it can be done faster because it does not need to redo

the functional simulation, but it needs to store large trace files and cannot simulate instructions executing along speculative paths. The main disadvantage of detailed simulation is that it is extremely slow (0.3 MIPS for sim-outorder, a detailed execution-driven simulator of the SimpleScalar toolset [4]). Even the fastest simulators running on today's fastest computers need several hours or days to simulate just a few seconds of real execution time. Moreover, since a detailed simulator has to model a very complex system, development time is long.

To overcome the huge evaluation time of detailed simulation while retaining an acceptable level of accuracy, several simulation speedup techniques have been proposed. Two main techniques are sampled simulation and statistical simulation. The goal of *sampled simulation* is to approximate the performance of the complete program by simulating only a small part of the program. If this part is representative for the whole program, accurate performance estimations can be made in much less time. To obtain accurate performance estimations, often more than one sampling unit per program is needed. Examples are random sampling [16], which randomly selects sampling units and uses statistical techniques to estimate the accuracy; SMARTS [101], which simulates small sampling units distributed periodically over the whole program execution; and SimPoint [42], which first profiles the program to find distinct execution phases, and than selects a sample such that every phase (or the main phases) is represented. Performance estimations are within a few percent of detailed simulation performance results, while the evaluation time is drastically reduced [102]. Since a detailed simulator is still needed, the development time is not reduced.

Statistical simulation [22, 37, 75, 76] also strives at minimizing the number of instructions to simulate, by building a synthetic program that has almost the same statistical characteristics as the original program, but that is several orders of magnitude smaller. This is done by first measuring different characteristics (by using a specialized functional simulator to obtain dependency information, cache and branch predictor behavior, etc.), and then use Monte Carlo simulation to build a small program trace with similar characteristics. Performance is estimated by simulating this synthetic trace. This technique yields good accuracy while needing much less time to evaluate performance (the functional simulations need to be done only once to enable the evaluation of a large range of processor configurations). The simulator needed to execute the synthetic trace is also less complex, because the synthetic trace captures some microarchitectural information (e.g., cache misses

and branch mispredictions). The development time is therefore smaller compared to detailed simulation.

The performance model elaborated in this thesis can be situated in the field of *analytical modeling*. Analytical modeling eliminates the detailed simulation by replacing it with simple formulas that provide insight. The inputs of the model are similar to that of statistical simulation—program dependency and locality characteristics (cache miss rates and branch misprediction rates), obtained from specialized functional simulation—but instead of building and simulating a synthetic program, performance evaluation is done by evaluating some mathematical formulas and algorithms, and is therefore extremely fast. Accuracy remains good, as will be shown in Chapter 4, and much less time is needed to develop an analytical model compared to developing a complex detailed simulator. Analytical modeling also provides more insight than simulation into the performance impact of different processor parts.

Chapter 3

Interval analysis

Divide et impera
(divide and rule)
Roman strategy to rule conquered nations

Analyzing the performance of a complex superscalar out-of-order processor is not trivial. It can have several hundreds of in-flight instructions, executing out-of-order. Different kinds of miss events and resource stalls can happen at different points in the processor pipeline, and miss events can overlap with each other. Therefore, it is obvious why a sufficiently detailed but still comprehensible model is difficult to build.

This chapter describes an intuitive and accurate model, called *interval analysis*, for analyzing superscalar processor performance. It is based on the observation that in the absence of miss events, a constant performance level is reached that is close to D instructions per cycle, with D equal to the designed processor width. Miss events cause performance to ramp down, leaving gaps in the constant performance level. In that sense, miss events divide total execution time into distinct intervals, and these intervals can then be studied in isolation. Using interval analysis, the performance impact of different types of miss events can be easily quantified, and a performance model can be developed.

In the first and second section we will define intervals by studying the dispatch behavior. In the next sections, different miss events are discussed and their performance impact is quantified.

3.1 Dispatch versus fetch, issue and commit

To study processor performance in terms of instructions handled per cycle, one can probe different points in the processor pipeline. Because every (correct-path) instruction has to be fetched, dispatched, issued and eventually committed, performance can be defined as the average fetch rate (excluding wrong-path instructions) as well as the average (correct-path instruction) dispatch, issue or commit rate, all resulting in the same number. Figure 3.1 shows parts of the execution profile of gcc, indicating the number of correct-path instructions handled per cycle for fetch, dispatch, issue and commit (dispatch, issue and commit are 4-wide, fetch is 8-wide and the front-end pipeline consists of 5 stages; see Appendix A for more details about the processor configuration).

In each graph the performance degradation due to miss events can be easily noticed. The different stages in the processor pipeline however respond differently to miss events. We now investigate which profile is best suited for building an easy-to-understand performance model that enables us to analyze the performance impact of miss events.

In the fetch stage, peaks are alternated with dips introduced by taken branches or instructions on different cache lines. These dips are not caused by miss events, but are due to the fact that the fetch engine is unable to fetch instructions from different cache lines in one cycle. To provide a constant flow of instructions to the rest of the processor front-end pipeline, these dips are offset by peaks, enabled by the fact that the fetch width is larger than the width of the other front-end pipeline stages. Only the larger parts where no instructions are fetched are due to real miss events. The occurrence of dips and peaks makes it difficult though to study performance at the fetch stage.

The dispatch stage shows an on / off behavior: performance goes up immediately after the resolution of the previous miss and drops quickly to zero after the next miss event. This makes it easy to define intervals and analyze them.

Issue on the other hand displays a less regular behavior. This is because the issuing of instructions is guided by the inter-instruction dependencies. Instructions have to wait until the instructions they depend on are executed. This makes the issue behavior harder to reason about. For example, when a miss event occurs, there may still be some unexecuted instructions in the instruction window, so issue can con-

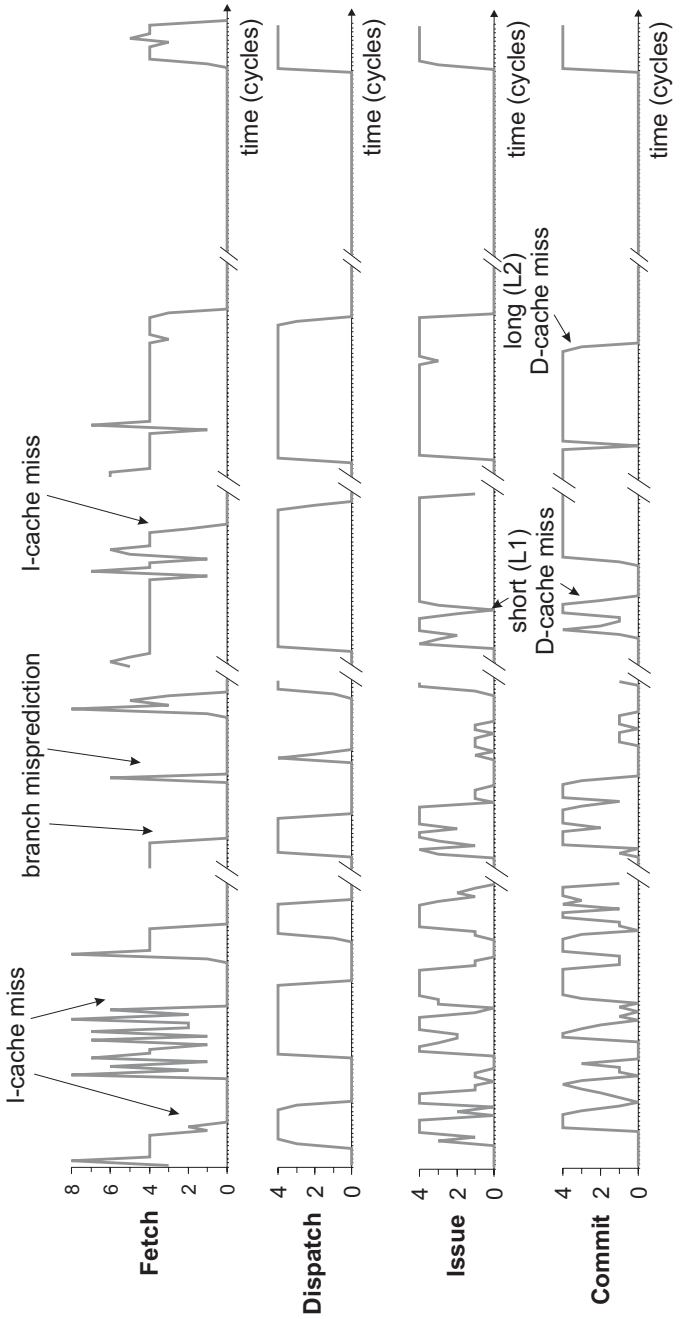

Figure 3.1: Parts of the execution profile of `gcc` executing on a superscalar processor in the fetch, dispatch, issue and commit stage.

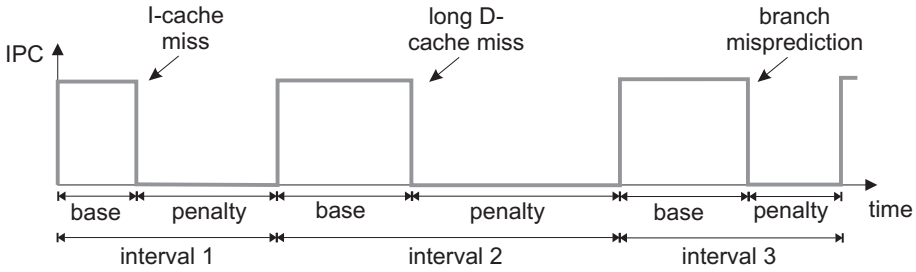

Figure 3.2: Dispatch behavior as an on / off function and definition of intervals.

tinue until these instructions are all executed. This means that the gaps in the issue rate introduced by miss events are less clear, making analysis more difficult. Commit eventually keeps up with issue, so it shows a similarly irregular behavior. Moreover, since commit is the very last stage in the processor pipeline, it is difficult to detect front-end miss event penalties, such as instruction cache misses.

We conclude that the dispatch stage shows the most regular behavior, which makes performance analysis easier, while being as accurate as studying other pipeline stages. The next section shows how to use dispatch behavior to construct a simple and accurate performance model.

3.2 Interval behavior

By studying dispatch behavior, it is easy to calculate penalties because performance is almost always either maximum or zero. Moreover, the dispatch stage is somewhere in the middle of the processor pipeline, close enough to both front-end and back-end miss events. In the following, we idealize dispatch behavior as an on / off function, see Figure 3.2. The inaccuracies introduced by this assumption will be discussed in Section 3.6.

For each type of miss event, we observe similar behavior: a miss event causes dispatch to stop, and when the miss event is resolved, dispatch resumes. An *interval* can now be defined as a slice of the total execution time that starts with the resolution of a previous miss event and ends just before the next miss event is resolved, see Figure 3.2. An interval therefore basically consists of two parts: the part before the miss event where performance is equal to D instructions per cy-

cle (with D the dispatch width), called the *base* part, and the part after the miss event where performance is zero due to the miss event, called the *penalty*. We define *interval length* as the number of (correct-path) instructions dispatched during the interval, while the term *interval* refers to the complete execution time slice (base + penalty).

Although interval behavior is similar for each type of miss event, the exact interval timing is dependent on the type of miss event. In the next two sections, the interval timing of each type of miss event will be discussed. The first section discusses front-end miss events (instruction cache misses and branch mispredictions) and the second section talks about back-end miss events (data cache misses and resource stalls).

3.3 Front-end miss events

Front-end miss events are miss events that occur before the dispatch stage. They cause a temporary gap in the supply of (correct-path) instructions into the instruction window. There are two main types of front-end miss events: instruction cache misses (and instruction TLB misses) and branch mispredictions. Because they show different behavior, they are separately discussed in the next two sections.

3.3.1 Instruction cache misses and instruction TLB misses

When an instruction is not present in the level-1 (L1) instruction cache, a lower level cache or even memory has to be accessed. This takes more time than a hit in the instruction cache and the constant supply of instructions is interrupted, for a time equal to the access time of that level in the memory hierarchy. The same happens when a virtual to physical page translation is not present in the instruction TLB, and has to be looked up in memory. Because the interval behavior of I-cache misses and I-TLB misses is similar, we consider instruction cache misses and instruction TLB misses together in this section.

Figure 3.3 is a detail of Figure 3.1 that zooms in on an instruction cache miss. When instruction fetch halts because of an instruction cache miss (1), the fetch buffer and front-end (decode) pipeline will drain (2). Then the dispatch of instructions will cease, causing issue and eventually commit to ramp down (3). When the miss is resolved, the fetching of instructions restarts, and the fetch buffer and front-end pipeline refills (4). Dispatch restarts when the first new instruction has passed

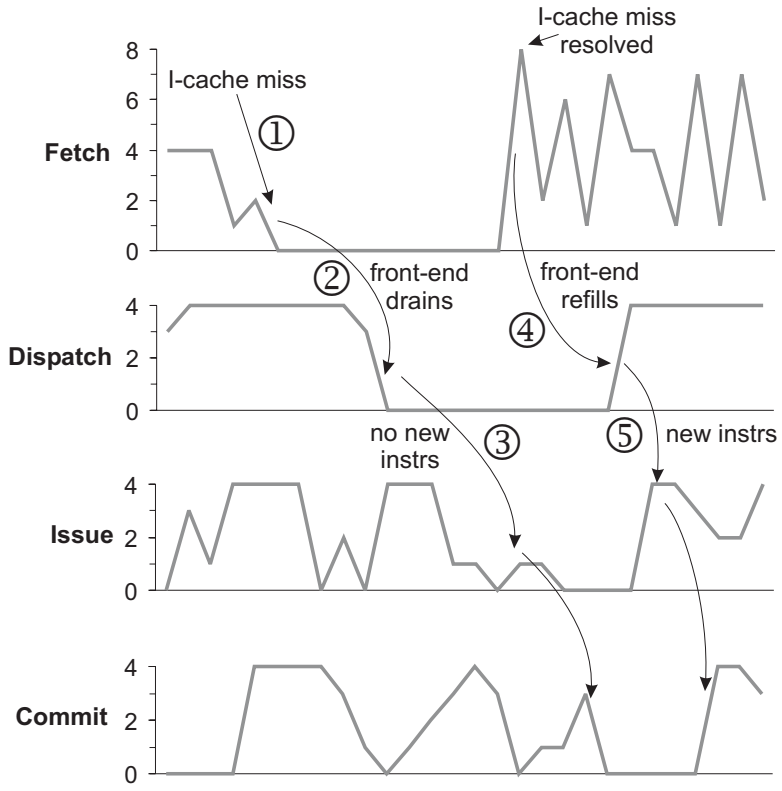

Figure 3.3: Behavior of an instruction cache miss.

through the whole front-end pipeline, and then issue and commit ramp up again (5).

The interval model of an instruction cache miss is displayed in Figure 3.4. As explained above, the penalty of a miss event equals the time dispatch is zero. In the case of an instruction cache miss, dispatch ramps down after the fetch buffer and front-end pipeline are drained, and ramps up after the miss is resolved, and the fetch buffer and front-end pipeline are refilled. Since the front-end pipeline drain and refill times offset each other, the total penalty equals the miss delay.

So the penalty for an instruction cache miss only depends on the cache or memory level that needs to be accessed, and is constant for all misses to the same level. This is experimentally validated in Figure 3.5, which shows the cycle penalty per L1 I-cache miss (and L2 cache hit) for different benchmarks. The penalty per L1 instruction cache miss is calculated as the extra number of cycles when simulating a real in-

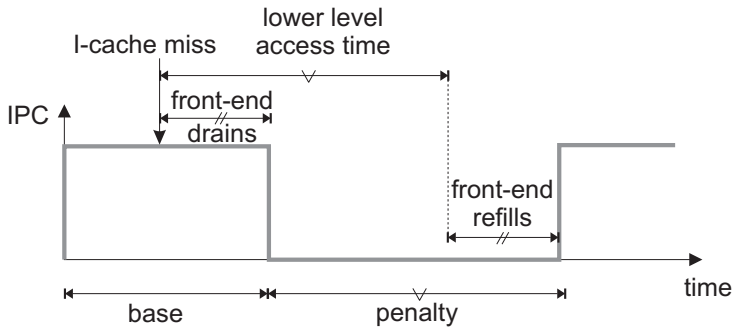

Figure 3.4: Interval timing of an instruction cache miss.

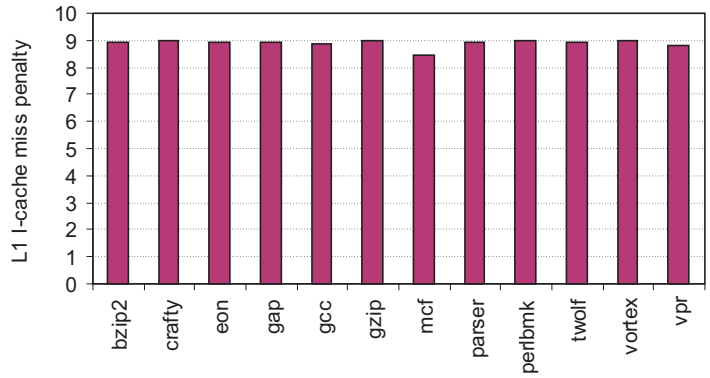

Figure 3.5: Average cycle penalty of an L1 instruction cache miss, assuming a 9-cycle L2 cache access time.

struction cache against a simulation with perfect cache behavior (all cache accesses are assumed to be cache hits), divided by the number of cache misses. The level 2 cache access time is assumed to be 9 cycles in this experiment. The fetch buffer causes the penalty to be somewhere between 8 and 9: if, at the moment of the cache miss, the fetch buffer contains more instructions than the dispatch width, there are some instructions left that can be dispatched in the next cycle, and one cycle of the I-cache miss penalty is hidden.

3.3.2 Branch mispredictions

A mispredicted branch does not stop the fetching of instructions explicitly, but guides instruction fetch in the wrong direction. The instruc-

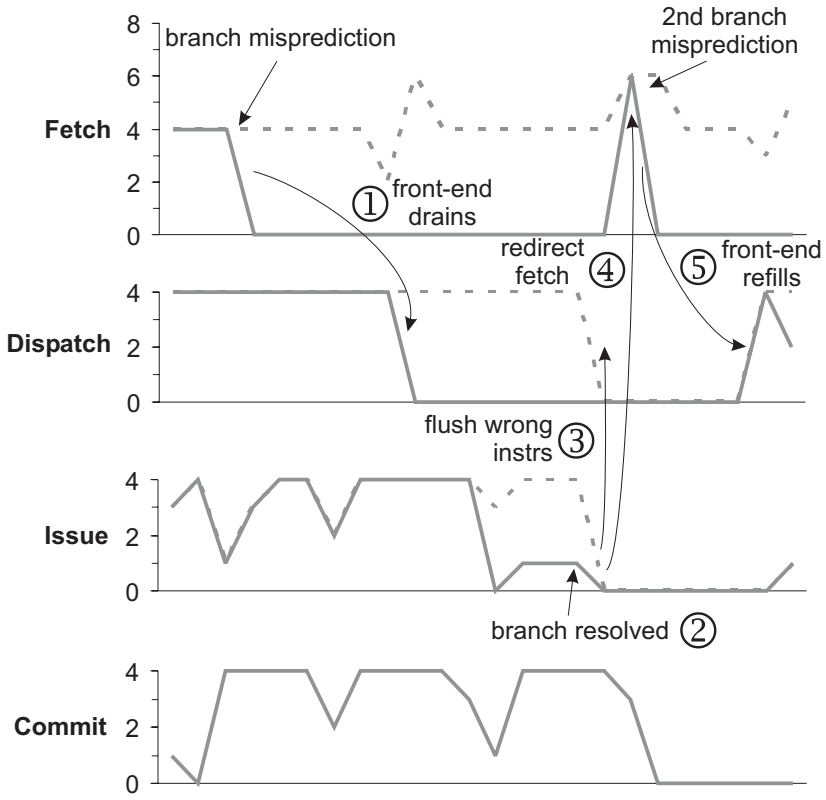

Figure 3.6: Behavior of a branch misprediction (the dotted line represents instructions along a mispredicted path).

tions fetched are along a wrong path, and will eventually be flushed. In that context, a branch misprediction can be seen as an inhibitor of fetching correct-path instructions, similar to an instruction cache miss. Figure 3.6 (again a detail of Figure 3.1) shows the effect of a branch misprediction at the fetch, decode, issue and commit stages. The dotted line indicates the fact that instructions are still fetched, dispatched and executed after a branch misprediction, but they are along a wrong path (and therefore are never committed). The solid line shows the progress of the correct-path instructions.

Similar to an instruction cache miss, the dispatch of useful instructions will cease after the correct-path instructions in the fetch buffer and front-end pipeline are drained (1). The last correct-path instruction dispatched is the mispredicted branch itself. When the branch is issued and executed (2), the misprediction is detected, wrong-path

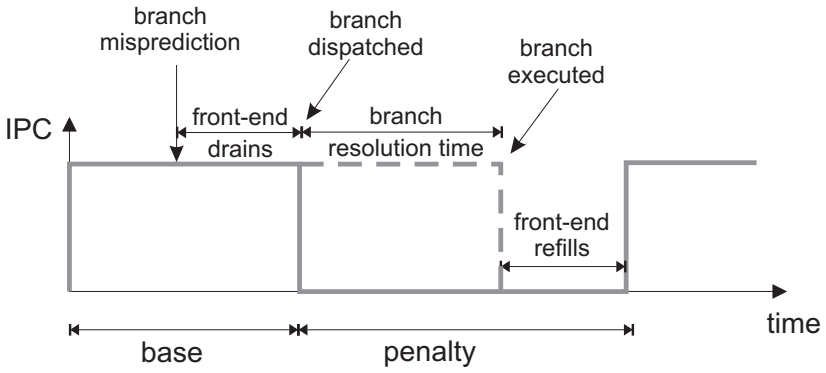

Figure 3.7: Interval timing of a branch misprediction.

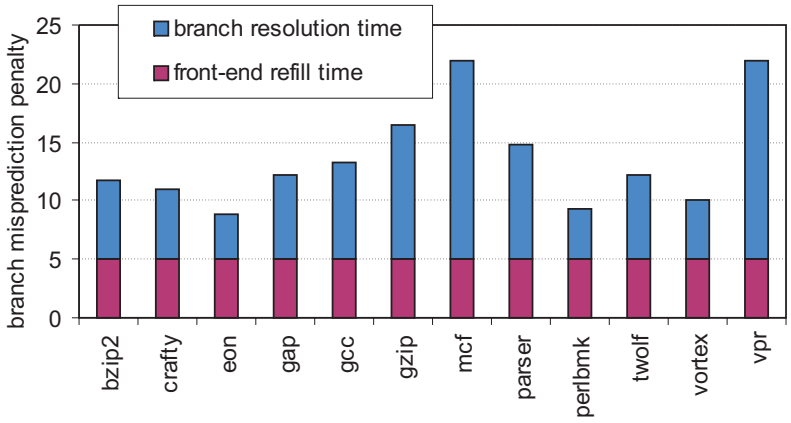

Figure 3.8: Average penalty of a branch misprediction.

instructions are flushed (3) and fetch restarts at the next correct-path instruction (4). Then the fetch buffer and front-end pipeline are filled with correct-path instructions and dispatch resumes (5).

Figure 3.7 depicts the interval timing of a branch misprediction. It shows how the penalty of a branch misprediction, i.e., the time between the dispatch of the branch and the dispatch of the next correct-path instruction, consists of two parts: (i) the time needed to execute the dependency path to the branch and the branch itself (the branch resolution time), and (ii) the time needed to refill the front-end pipeline.

Figure 3.8 shows the average branch misprediction penalty for each benchmark (calculated by comparing a simulation with perfect branch

prediction to a simulation with a realistic branch predictor, as done for determining the penalty per instruction cache miss in the previous section). The front-end pipeline depth is assumed to be 5 stages in this experiment. These results show that the actual branch misprediction penalty can be much larger than the front-end pipeline depth, which is often assumed to be the penalty of a branch misprediction. The penalty also includes the time needed to execute the dependency path to the branch. In the next chapter, we will explain how the branch resolution time is affected by program characteristics, i.e., the interval length distribution, the average critical dependency path length and the average instruction execution latency.

3.4 Back-end miss events

Back-end miss events are data cache or data TLB misses. Because they occur in the back-end of the processor, they exhibit a completely different behavior compared to front-end misses. In fact they only enforce dispatch to ramp down if they cause a resource stall, i.e., a processor component becomes full and accepts no new instructions. We make a distinction between long data cache misses, i.e., data has to be fetched from memory, which take long enough to always cause a penalty, and short data cache misses and other resource stalls, which occasionally incur short penalties [54].

3.4.1 Long data cache misses

Main memory is nowadays two orders of magnitude slower than the processor core. Data cache misses that have to access memory take hundreds of processor cycles. Stores that cause a data cache miss have a limited performance impact, because they do not produce a result needed by the processor, which means they can be buffered without stalling the processor—only when the store buffer is full, a store miss will cause the processor to stall, but this happens relatively infrequently. Load misses on the contrary have to wait the complete memory access time in the processor before they can be finished and committed.

An important difference compared to front-end misses, is that back-end misses can overlap with each other. We will first discuss the in-

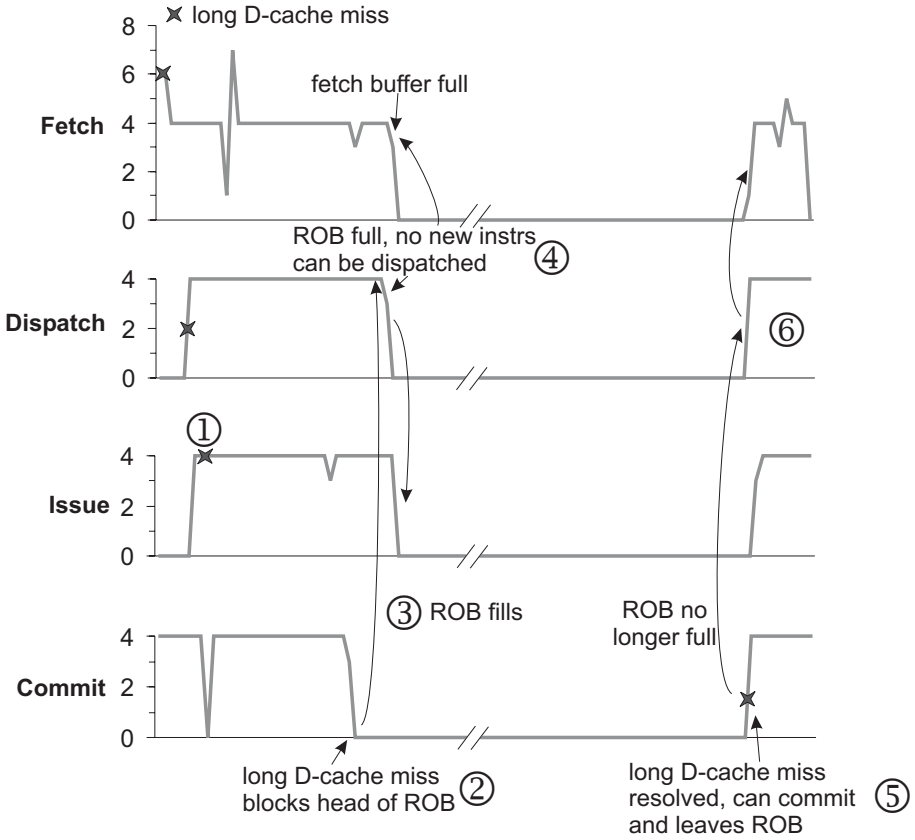

Figure 3.9: Behavior of a long-latency load miss.

terval behavior of one long-latency load miss in isolation, and subsequently treat overlapping load misses.

Isolated long-latency load miss

The impact of a long-latency load miss is shown in Figure 3.9 (taken from the execution profile of gcc). A long-latency load miss (or a data TLB load miss) is detected when it is issued (1) and the load has consulted all the cache levels. Then it has to wait until the data returns from memory. When all preceding instructions are finished and committed, the load miss comes at the head of the ROB, and blocks the commit stage until it is resolved (2). So commit will ramp down first, and since instructions keep entering the ROB, it will fill up (3). At a certain point, the ROB will be full, and no new instructions can be dispatched (4).

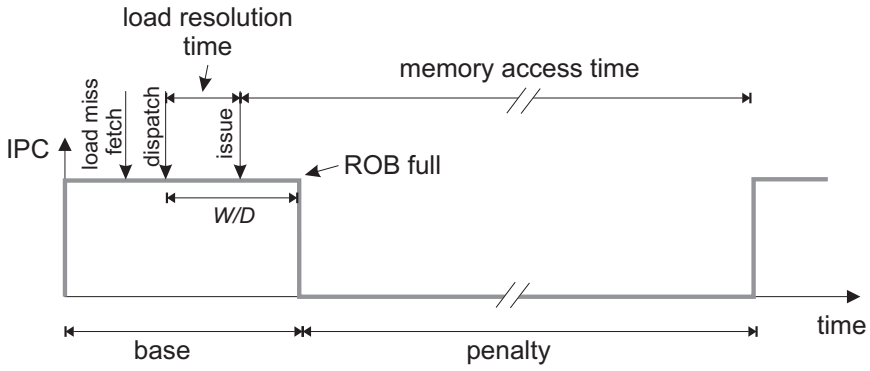

Figure 3.10: Interval timing of a long-latency load miss.

At that moment dispatch stops, which causes fetch to stall because the front-end pipeline cannot be drained and also fills up. Instructions that are independent of the load miss can still issue, but soon there will be no such instructions available, and issue ramps down.

When the load miss returns from memory, it can finally commit (5), and ROB entries are freed. New instructions can be dispatched from the front-end, reactivating the fetch stage (6). Instructions that were dependent on the load miss can now be issued and new instructions enter the instruction window, so issue also ramps up.

To calculate the penalty, we should again look at how many cycles dispatch is stalled. The discussion is now more complicated than for front-end misses, see Figure 3.10. Dispatch stops when the ROB is full. This happens when there are W (ROB size) instructions dispatched (including the load miss) after having dispatched the load instruction that caused the miss. This takes W/D cycles. Dispatch restarts when the load miss returns from memory. This takes a number of cycles equal to the memory latency (including the time to consult all cache levels) after issuing the load. The time between the dispatch and issue of the load is the resolution time, i.e., the time needed to execute all instructions the load depends on after dispatching the load. So the total penalty of a long latency load miss can be calculated as the resolution time plus the memory latency minus W/D.

This means that the penalty of a long-latency load miss depends on (i) the ROB size W and the dispatch width D—which is fixed for a certain processor; (ii) the main memory access latency—which is also approximately constant (there can however be some variance through bus

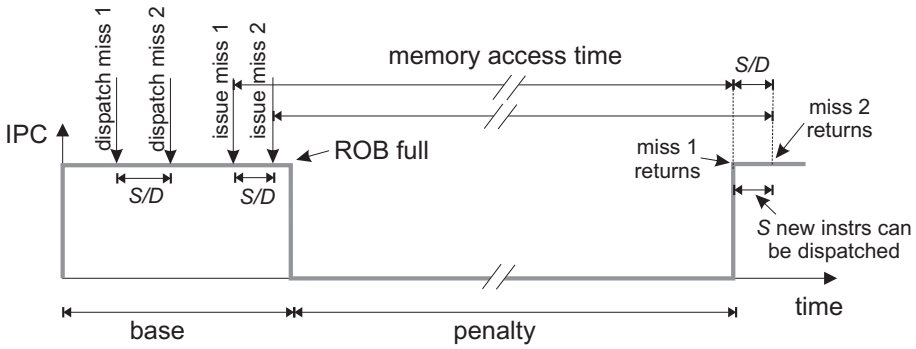

Figure 3.11: Timing of two overlapping long latency load misses.

contention and memory bank conflicts); and (iii) the resolution time of the load. The latter depends on the number of instructions that need to be executed before the load can issue, which is dependent on the application running. However, considering that the main memory latency is in the order of hundreds of cycles, and the resolution time is usually not more than about ten cycles, the impact of this variance is rather small. For moderate ROB sizes, the W/D term is also rather small compared to the memory latency. Furthermore, the W/D term and the load resolution time have opposite signs in the penalty calculation, which means that they partially offset each other. This lets us conclude that the penalty for an isolated long data cache load miss is nearly constant, and is approximately equal to the main memory access latency.

Overlapping long-latency load misses

When a load miss is dispatched, the dispatch of new instructions continues until the ROB is full. These new instructions can include another long-latency load miss. If this long-latency load miss is independent of the first miss, it will be issued, and its waiting time will start concurrently with the penalty of the first load miss (assuming that the memory can handle multiple accesses in parallel[1]). So the penalties of the two misses will overlap.

To quantify the amount of overlap, assume there are S instructions (in the dynamic program trace) between the two load misses, which means the second miss will be dispatched S/D cycles after the first, see

[1]This can be achieved by the use of miss status holding registers (MSHRs) and memory banks.

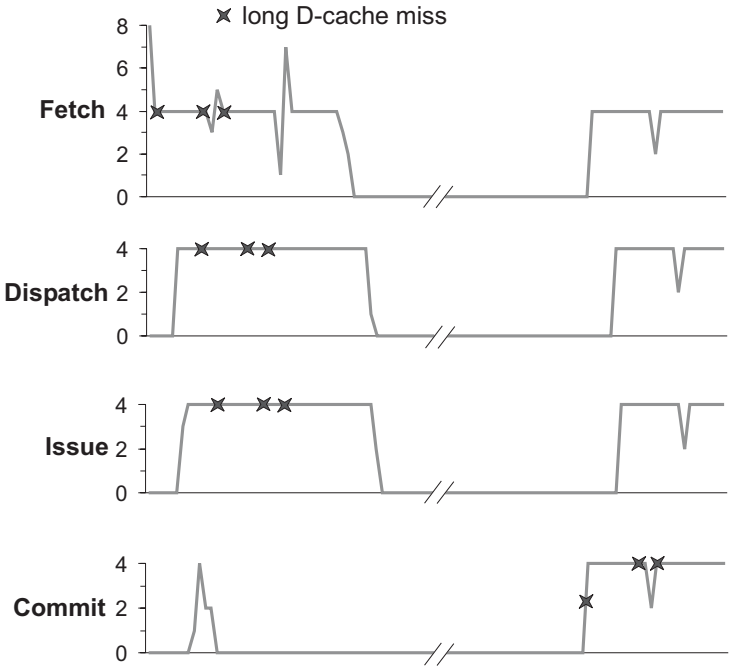

Figure 3.12: Behavior of three overlapping load misses.

Figure 3.11. If the resolution time of the two misses is almost equal, the time between issuing the misses is also S/D. This means that the data returning from memory also happens S/D cycles from each other. If the first miss commits, S instructions can commit until we meet the second miss at the head of the ROB, leaving space for the dispatch of S new instructions. If the commit width is at least as large as the dispatch width (which is usually the case), then these S new instructions can be dispatched at the full width, needing S/D cycles, which is exactly equal to the time until the second load miss returns from memory. So by the time dispatch should stall because of the second load miss hitting the head of the ROB, the second load miss commits and dispatch can continue. The result is that the two misses completely overlap, i.e., the penalty for the second miss is completely hidden under the penalty of the first one. This can be generalized to three or more independent load misses that can start their execution before the ROB is full, i.e., their penalties completely overlap, and the total penalty equals the penalty of one isolated load miss.

To illustrate this effect, Figure 3.12 shows a part of the execution

profile of gcc where three long-latency load misses overlap. After the first load miss returns from memory (indicated in the commit stage), performance ramps up again, and there is almost no performance degradation due to the following load misses. There is however a small decline in performance between committing the second and third load miss. This is due to the fact that these load misses arrived closely together, causing some (L2 to memory) bus contention. The result is that the latency of the second load miss is somewhat larger (one cycle in this example), which violates the assumption of a constant memory penalty in the reasoning above. This performance decline is however negligible compared to the memory access latency.

This means that all independent load misses that occur at most W (ROB size) instructions after a load miss overlap with the first one, because they all enter the ROB before it is full and handle their latencies in parallel. This effect is called *memory-level parallelism (MLP)* [14, 38]. MLP can be quantified as the average number of long-latency load misses that are handled in parallel if there is at least one such miss outstanding. Considering the previous reasoning—i.e., the fact that parallel load misses completely overlap—this can be simplified to the total number of load misses divided by the number of non-overlapping misses. It gives an idea by how much the penalty of each load miss is hidden through memory-level parallelism. If the MLP equals 1, all load misses see their full penalty and no memory-level parallelism is exposed. If it is greater than 1, on average MLP load misses overlap, and the total penalty is shared between these load misses (e.g., an MLP of 2 means that the penalty of an isolated load miss is shared between on average 2 load misses, so the average penalty per load miss is half the penalty of an isolated load miss).

Figure 3.13 depicts the average number of cycles per load miss (calculated as the extra number of cycles of a simulation with a real L2 cache compared to a simulation with a perfect L2 cache, divided by the number of long-latency load misses). It shows that for some benchmarks the average load miss penalty is close to the memory access time (250 cycles in this experiment) (e.g., mcf, perlbmk and vortex), but other benchmarks expose much lower penalties due to a large degree of MLP (e.g., crafty, eon and gzip).

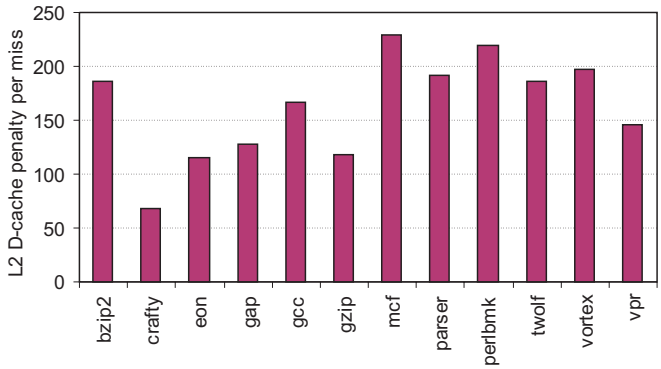

Figure 3.13: Average cycle penalty of a long data cache miss (memory access latency is 250 cycles).

3.4.2 Other resource stalls

Short data cache misses, i.e., those that are handled by the second level cache, also make a load waiting for data. This additional latency is however rather short (at most some tens of cycles), and they usually are resolved before the ROB fills up and dispatch is stalled. This means they incur no, or little, penalty because they are typically hidden through out-of-order execution. The same happens with other medium-latency instructions, such as (floating-point) multiply and divide instructions.

In some cases there is a long dependency path leading to the short data cache miss or medium-latency instruction, which causes the resolution time to be long. If the sum of the resolution time and the instruction's execution latency is larger than the ROB fill time, the ROB will fill up, ceasing dispatch for a few cycles. In that case, there is a small penalty, but on average, the penalty of short data cache misses is very small or negligible. This is confirmed in Figure 3.14, where the cycle penalty per L1 data cache miss that is served by the L2 cache is illustrated (L2 cache access time is 9 cycles). Only for parser, twolf and vpr, the penalty per miss is somewhat higher (but still much lower than the L2 access time), due to the long dependency paths.

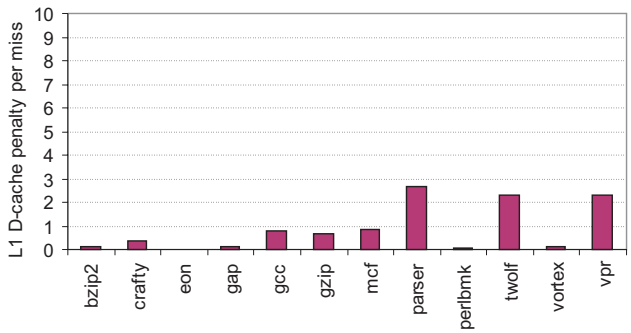

Figure 3.14: Average cycle penalty of a short data cache miss (L2 cache access latency is 9 cycles).

3.5 Overlapping miss events

In the previous sections we have considered miss events in isolation. In a real execution, miss events can come close to each other, and their penalties can partially or completely overlap, which reduces the total penalty seen by the processor. The previous section already discussed overlapping long-latency load misses. We now discuss overlap effects among different types of miss events. We make a distinction between possible overlaps between two or more front-end misses, and front-end miss events overlapping back-end miss events or vice versa.

3.5.1 Overlaps between front-end miss events

Instruction cache misses cannot overlap other front-end misses, because they interrupt the supply of new instructions, and front-end miss events can only be caused by new instructions. In the case of a branch misprediction, instruction fetch continues along the mispredicted path. Along this path, instruction cache misses or new branch mispredictions can occur, but after the detection of the misprediction, these instructions are flushed and fetch is redirected to the correct path, so these extra miss events have no direct impact on the total performance. There can be an indirect impact on performance, especially when the mispredicted path reconverges with the correct path (e.g, the instructions after an if-then-else clause). In this case the instructions along the mispredicted path will be re-executed on the correct path, and

the first execution can serve as a (instruction and data) prefetcher for the second execution. Ignoring these second-order effects, we can conclude that front-end misses along mispredicted paths can be ignored, which means that there are no mutual overlaps between front-end miss events.

3.5.2 Overlaps between front-end and back-end miss events

When dealing with a long-latency load miss, new instructions can still be fetched for a while, and these new instructions can introduce front-end misses. We now discuss possible overlaps between long-latency load misses on the one hand, and instruction cache misses and branch mispredictions on the other hand.

Overlaps between long-latency load misses and instruction cache misses

The situation of a *short instruction cache miss* that occurs when a long-latency load miss is pending is illustrated in Figure 3.15. When the long-latency load miss issues (1), dispatch can still continue until the ROB is full. If a short (L1) instruction cache miss occurs before the ROB is full, it will cease dispatch (2) until the instruction has been fetched from the L2 cache and the front-end pipeline has been refilled (3). This takes much less time than the long-latency load miss, so dispatch restarts long before the load miss returns from memory. The ROB will continue to fill up, and when it is full (4), dispatch stalls until the load miss returns from memory (5). The penalty of the short instruction cache miss is thus completely overlapped by the long-latency load miss, because the instructions cache miss is being resolved long before the load miss returns.

In case of a *long instruction cache miss* (i.e., the instruction has to be fetched from memory), the ROB will never fill up, because there are no new instructions to enter the ROB during the whole latency of the load miss (since the load miss and the instruction cache miss have approximately equal latencies), see Figure 3.16. This means that the penalty of the load miss is completely overlapped by the long-latency instruction cache miss penalty.

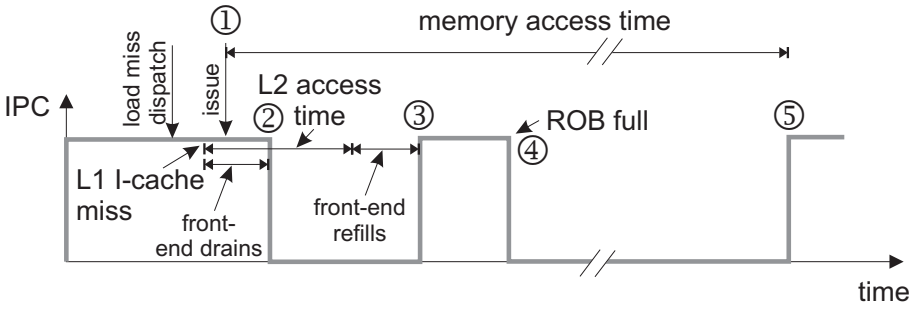

Figure 3.15: Overlap between a long data cache miss and a short instruction cache miss.

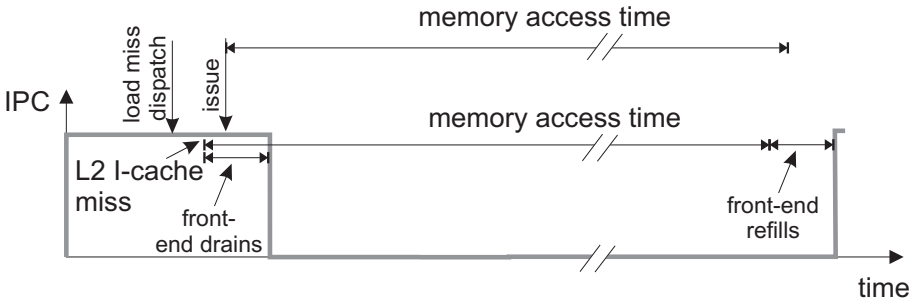

Figure 3.16: Overlap between a long data cache miss and a long instruction cache miss.

Overlaps between long-latency load misses and branch mispredictions

When a branch misprediction occurs less than W instructions after the dispatch of the long-latency load miss, two things can happen. If the branch is *independent* of the load miss, it will be resolved before the load miss returns, see Figure 3.17. This means the penalty of the branch misprediction is completely overlapped by the penalty of the long-latency load miss, similar to the overlap between a long-latency load miss and a short instruction cache miss.

If a branch that is *dependent* on the long-latency load miss is mispredicted while handling the load miss, it will not be detected until after the data returns from memory. This situation is depicted in Figure 3.18. The branch misprediction causes the dispatch of wrong-path instructions (1). These instructions will also fill up the ROB (2). After the load miss returns from memory (3), dependent instructions are

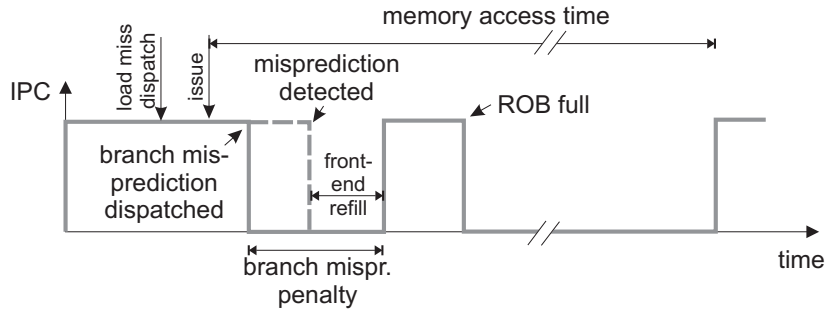

Figure 3.17: Overlap between a long data cache miss and an independent branch misprediction.

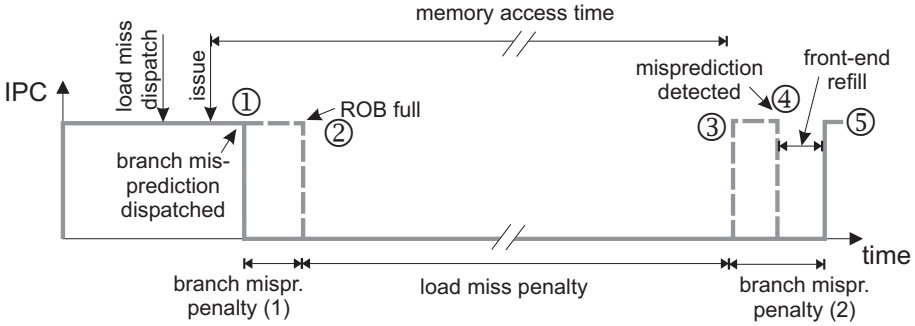

Figure 3.18: Overlap between a long data cache miss and a dependent branch misprediction.

executed until the branch miss is detected (4). During this time, wrong-path instructions keep on dispatching, because the branch misprediction is not yet detected. After the detection of the branch misprediction, wrong-path instructions are flushed, and the front-end pipeline needs to be refilled to restart dispatch (5). The total penalty is the gap indicated by the solid line and includes the total load miss penalty (i.e., the time the ROB is full) plus the total branch misprediction penalty (i.e., the time where wrong path instructions are dispatched, plus the front-end pipeline refill time). The two penalties are thus serialized instead of parallelized.

Impact of overlaps between front-end and back-end misses

Front-end misses that are overlapped by back-end misses can involve difficulties for analyzing and estimating performance. In order to have

an accurate model these overlaps should be modeled, which complicates the model. Table 3.1 shows the fraction of total execution time where such overlaps occur, as a function of the ROB size. This is the percentage of cycles where the penalties of instruction cache misses and branch mispredictions are completely hidden under the penalty of a long-latency load miss. It is an indication for the overestimation of the execution time when assuming that no overlaps occur and all penalties serialize. Overlaps occur more often as the ROB size enlarges because more instructions can be fetched under the long-latency load miss. This table also shows that for moderate ROB sizes, overlaps are rather rare (less than 5.5% for a ROB of 256 instructions). This means that in order to build an accurate model for realistic processor architectures, neglecting these overlaps introduces only a small error.

Table 3.1: Percentage cycles for which front-end penalties are overlapped with long back-end miss penalties for different ROB sizes.

Benchmark	ROB 64	ROB 128	ROB 256	ROB 512	ROB 1024
bzip2	0.03%	0.07%	0.15%	0.29%	0.37%
crafty	0.13%	0.37%	0.75%	0.87%	0.80%
eon	0.00%	0.01%	0.02%	0.03%	0.03%
gap	0.36%	1.62%	5.40%	15.97%	18.80%
gcc	0.15%	0.31%	0.48%	0.65%	0.67%
gzip	0.00%	0.01%	0.03%	0.05%	0.08%
mcf	0.00%	0.00%	0.02%	0.06%	0.06%
parser	0.11%	0.31%	0.72%	1.39%	1.75%
perlbmk	0.02%	0.05%	1.77%	5.72%	13.73%
twolf	0.63%	1.64%	3.44%	4.50%	4.91%
vortex	0.30%	0.99%	1.87%	2.97%	3.78%
vpr	0.64%	1.65%	3.23%	4.92%	4.83%

3.6 Inaccuracies due to the dispatch on/off behavior assumption

In the timing analysis above we always assumed that the dispatch rate is an on/off process, i.e., the number of instructions dispatched per cycle is either zero or equal to the designed dispatch width. As can be seen from the gcc example (Figure 3.1), this is a good approximation, but in some cases the dispatch rate is less than D instructions per cycle. There are two possible reasons:

1. *Inherent dispatch inefficiency.* Interval lengths are not always a multiple of the dispatch width. This means that in the last cycle of an interval where instructions are dispatched, there are possibly less than the available width instructions left to be dispatched until the miss event occurs. This dispatch inefficiency is inherent, it is caused by the occurrence of miss events and depends on the dispatch width of the processor—the larger the dispatch width, the larger is the probability that the interval size is not a integer multiple of the dispatch width. The performance estimation method elaborated in the next chapter takes these inefficiencies into account.

2. *Fetch inefficiency.* Due to taken branches, the fetch engine is not always able to fill the fetch buffer with at least as many instructions as the dispatch width. In this case, fewer instructions can enter the front-end pipeline and will eventually be dispatched. This often occurs at the beginning of an interval, when the fetch buffer is empty, and one of the first instructions is a taken branch (or is at the end of a cacheline). However, in steady-state, a well designed fetch buffer should always provide enough instructions to sustain the designed dispatch width.

3.7 Summary

In this chapter, we have introduced *interval analysis* to analyze the performance of out-of-order superscalar processors. It is based on studying the dispatch behavior of a program executing on a processor, since this stage shows an on / off behavior, which makes it intuitive to analyze the performance impact of miss events. Using this approach, we can divide total execution time into isolated intervals, delimited by miss events. Each interval consists of a base part, where instructions are dispatched at the full designed width, and a penalty part, where the dispatch rate is zero due to the miss event.

The penalty of each type of miss event is composed differently.

- The *instruction cache miss* penalty equals the time needed to fetch the instruction from the appropriate level in the memory hierarchy, i.e., the L2 cache access time for a L1 instruction cache miss and the memory access time for a L2 miss. The penalty of an

I-TLB miss equals the time needed to consult the page table in memory.

- The *branch misprediction* penalty consists of the branch resolution time, i.e., the time needed to execute the dependency path leading to the branch, plus the front-end refill time. Due to the branch resolution time, the total branch misprediction penalty can be much larger than the front-end pipeline depth, which is often assumed to be the penalty of a branch misprediction.

- The penalty of an (isolated) *long-latency load miss* is equal to the load resolution time (the time between dispatching and issuing the load miss) plus the memory access time minus W/D. Since the memory access time is much larger than the other two terms, the long-latency load miss penalty can be approximated by the memory access time.

- *Short data cache misses* and *medium-latency instructions* usually do not result in a performance penalty, because their latencies are hidden by out-of-order execution. Only if they are on long dependency paths, they can fill up the ROB and introduce a small penalty.

We also discussed how long-latency load misses can be handled in parallel, and showed that the penalties of two or more concurrent independent long-latency load misses completely overlap with each other. This phenomenon is called *memory-level parallelism (MLP)*. MLP is defined as the average number of long-latency load misses that are handled in parallel, and quantifies how many load misses on average share the penalty of one isolated long-latency load miss.

Overlaps between front-end and back-end of miss events are rather infrequent for modest ROB sizes, but may need to be considered for larger ROB sizes.

Chapter 4

Mechanistic performance model

The inside of a computer is as dumb as hell
but it goes like mad!
Richard Feynman

Interval analysis as discussed in the previous chapter describes the cycle penalty of each miss event in an intuitive way. We now use these insights to construct a performance estimation model. The model estimates the number of cycles needed to execute an application by counting the number of miss events and estimating the penalty of each miss event. Adding the miss event penalties to the number of base cycles (i.e., the number of cycles needed to dispatch all instructions of the program) yields an estimation of the total cycle count. Performance in terms of IPC (instructions per cycle) can then be calculated as the number of instructions divided by the estimated cycle count.

This simple model assumes that only miss events introduce penalties. This is the case in balanced processor designs, i.e., all processor resources are sized such that the attained issue rate in the absence of miss events is (almost) equal to the designed issue width. In the context of out-of-order superscalar processors, this means that the instruction window (and its related structures, such as the issue buffer, the physical register file, etc.) is large enough to provide each cycle at least as many independent instructions as the processor's issue width. A method to characterize the available instruction-level parallelism (ILP)

in an application and how to scale processor resources to obtain a balanced design is described in the Section 4.1.

Section 4.2 then shows how the performance model is built up, and validates its accuracy in comparison to detailed simulation. Section 4.3 qualitatively compares the model to previously proposed methods for modeling the performance of out-of-order architectures. Section 4.4 uses the model to study pipeline depth and width, and deduces an interesting relationship between optimal pipeline depth and width, i.e., optimal pipeline depth and width are in opposition to each other: if the pipeline width increases by a factor c, the optimal pipeline depth decreases with a factor $\sqrt{c}$.

4.1 ILP characterization and balanced design

The discussion of interval analysis in the previous chapter assumed that dispatch only degrades through miss events. In some cases however, the instruction window fills up without having a miss event. This can happen if the amount of ILP that can be extracted from the instruction window is smaller than the dispatch width. This may cause the instruction window to eventually fill up, which causes dispatch to degrade. In this case, the processor is unbalanced, meaning that the instruction window should be scaled up in order to provide more parallelism, or, reversely, the processor is designed too wide for this instruction window size. In this section, we will describe an analytical method to measure the available instruction-level parallelism in a program and scale processor resources such that it is balanced.

Consider a window of W instructions and a given processor width D. In order to have on average D independent instructions to execute each cycle, a processor should be able to execute all instructions in the window in W/D cycles (Little's law). This implies that the critical dependency path, i.e., the longest dependency path within the instruction window, has to take at most W/D cycles to execute. Inverting this argumentation, we can say that a window W with a critical dependency path that takes K cycles to execute, provides on average W/K independent instructions each cycle, which means that it can support a balanced processor width of W/K.

To characterize the available ILP in a program and provide data to design a balanced processor, we should therefore study the critical dependency path of that program within a certain window size. We have

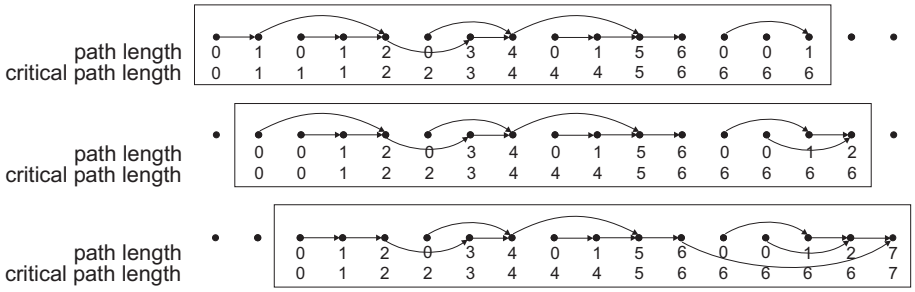

Figure 4.1: Calculation of the critical dependency path.

developed an algorithm that measures the average critical dependency path for all window sizes concurrently (from size 1 up to some maximum size W), using a sliding window that covers all possible windows of consecutive instructions of the dynamic program trace.

To do this, we start with analyzing the first window of instructions using a specialized functional simulator, see Figure 4.1. We keep track of the (register and memory) dependencies between the instructions, and calculate the length (in number of instructions) of the longest dependency path in the window. This can be done by going through the instructions in program order, initializing independent instructions with a zero dependency path, and, for instructions that depend on other instructions, adding one to the longest dependency path of all instructions it depends on. The maximum dependency path seen so far is retained, which is equal to the critical dependency path in the given window.

By retaining the critical dependency path length from the head of the window up to every instruction in the window (see Figure 4.1, in the row indicated by critical dependency path length), we can determine the critical dependency path length of all window sizes between 1 and the maximum window size in one sweep. For example, the top part of Figure 4.1 shows that the critical dependency path length for window sizes 2 to 4 equals 1, for window sizes 5 and 6 the critical dependency path consists of 2 instructions, for window size 7 the critical path length equals 3, etc.

When the maximum window size is reached, we shift the window one instruction, by removing the oldest instruction and all of its dependency relations, and adding a new one, calculating the new dependency relations for that instruction. Then we recalculate the depen-

Table 4.1: Power law estimates of $K(W)$ as a function of α and β; benchmarks are sorted by increasing β.

benchmark	α	β
vpr	1.40	1.31
perlbmk	1.55	1.33
parser	1.59	1.36
twolf	1.26	1.47
gap	1.34	1.50
bzip2	1.46	1.53
gzip	1.16	1.60
mcf	1.35	1.61
eon	1.51	1.66
vortex	1.44	1.67
gcc	1.67	1.68
crafty	1.06	1.79

dency paths for the new window. The complexity of this algorithm is $O(N \cdot W)$, with N the number of dynamic instructions of the program, and W the maximum window size.

The average critical dependency path for window sizes up to 1K instructions is shown in Figure 4.2. The graphs show approximately a straight line on a log-log scale, indicating that there is a power law between the window size W and the average critical dependency path K, i.e.,

$$K(W) \cong 1/\alpha \, W^{1/\beta}.$$

The parameters α and β are program dependent, and can be found by fitting the measurements to the power law, see Table 4.1.

Average critical path length has previously been used to characterize ILP by Michaud et al. [68]; there it is assumed that $\beta = 2$, i.e., a square root relation between the window size and the available parallelism. This is a rough approximation, as we observe β-values between 1.3 and 1.8, see Table 4.1. The higher the value of β, the shorter the critical path, and the more ILP is present; β is thus a measure of the inherent ILP in a program.

The minimum execution time of a window of instructions is the sum of the execution latencies of all instructions along the critical path in that window. To estimate the critical path length with non-unit execution latencies, we calculate the average instruction latency of all dynamically executed instructions. We do this by profiling the instruction

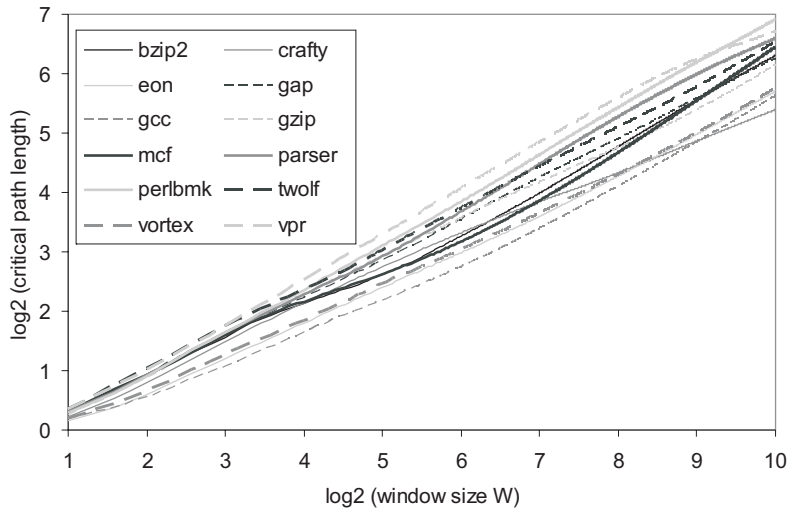

Figure 4.2: Average critical dependency path $K(W)$ as a function of window size.

type mix and by weighing the occurrence frequencies with the appropriate execution unit latencies. We then multiply the average instruction latency by the average critical path length. By doing so, a recalculation of the critical dependency paths and the instruction type mix is not needed when the instruction execution latencies change. The average critical dependency path and instruction type mix are properties of the application, while the execution unit latencies are dependent on the microarchitecture. Keeping these apart enlarges the portability of the model and minimizes the recalculation time when the instruction execution latencies are changed.

So, the average minimum execution time of a window W can be calculated as

$$\ell K(W) = \ell/\alpha \, W^{1/\beta},$$

with ℓ the average instruction execution latency. This means that the average number of independent instructions that can be found in an instruction window of size W is

$$I(W) = \frac{W}{\ell/\alpha \, W^{1/\beta}} = \alpha/\ell \, W^{1-1/\beta}$$

(see Figure 4.3). Using the approximation $\beta \approx 2$, we get

$$I(W) = \alpha \frac{\sqrt{W}}{\ell},$$

which reflects the well-known square root relationship between window size and issue rate [68, 81, 100].

We can also invert the $I(W)$ equation to get an expression that shows how the balanced instruction window size relates to the processor width: for a program with critical dependency path parameters α and β, in order to be balanced, a processor with width D should contain an instruction window of size

$$W(D) = (\ell D/\alpha)^{\beta/(\beta-1)}.$$

This means that the balanced window size is proportional to the dispatch width and the average instruction execution latency. Using the approximation $\beta \approx 2$, we get

$$W(D) = (\ell D/\alpha)^2,$$

which means that the balanced window size scales quadratically with both the dispatch width and the instruction execution latency. For β between 1.3 and 1.8, the power of the above formula ranges from 2.3 to 4.3.

The fact that our model assumes a balanced processor seems limiting, but from the critical path data we obtained, we can conclude that even for programs that show the longest critical dependency paths (e.g., vpr and perlbmk) a window of 1K instructions is sufficient to support a processor (issue) width of 8 (see Figure 4.3), which is wider than most of the current general-purpose processors. For certain applications or execution phases of an application, this balance condition may be violated, but the regular occurrence of miss events that drain the instruction window prevents the error from being propagated.

4.2 Estimating performance

As described in the beginning of this chapter, performance can be estimated by adding the miss event penalties to the base cycles. The latter could be easily calculated as the number of instructions divided by the dispatch width, but due to the fact that interval lengths are not always

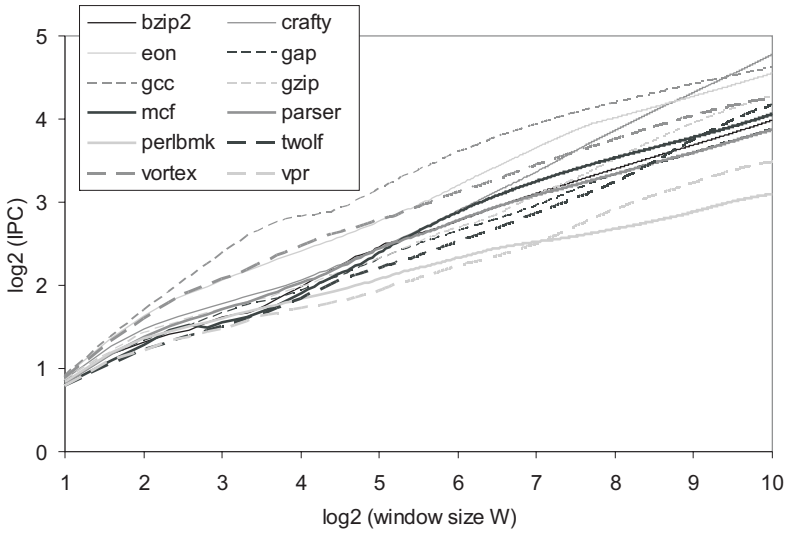

Figure 4.3: Average number of independent instructions $I(W)$ as a function of window size.

a multiple of the dispatch width, there is an inherent dispatch inefficiency, discussed in Section 4.2.1. In Section 4.2.2 it is shown how the different miss penalties are estimated, and Section 4.2.3 discusses the overall model. Section 4.2.4 then evaluates the accuracy of the overall model.

4.2.1 Inherent dispatch inefficiency

As discussed in the previous chapter, in the last dispatch cycle less than D instructions may be dispatched[1]. The reason is that the interval length (i.e., the number of instructions in the interval) is not a multiple of the dispatch width. The time needed to dispatch an interval of N_i instructions is $\lceil N_i/D \rceil$, and the total base time for the entire program execution is $\sum_{i=1}^{m} \lceil N_i/D \rceil$ (with m the number of intervals, which equals the number of miss events), which is not equal to $\frac{1}{D} \sum_{i=1}^{m} N_i$.

The additional number of cycles, introduced by the ceiling function, can be estimated as follows: assume that $N_i \bmod D$ is uniformly distributed, which means that $\lceil N_i/D \rceil - N_i/D$ is uniformly distributed

[1] In the following, we ignore the impact of fetch inefficiencies at the beginning of an interval, see Section 3.6.

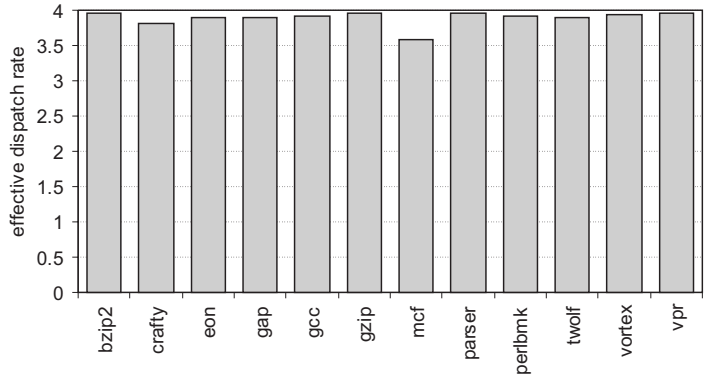

Figure 4.4: The effective dispatch rate is lower than the maximum dispatch rate of 4 because of interval behavior.

in $\{0, 1/D, 2/D, \ldots, (D-1)/D\}$. The mean of this distribution is $\frac{D-1}{2D}$, so

$$\sum_{i=1}^{m} \lceil N_i/D \rceil \cong \frac{\sum_{i=1}^{m} N_i}{D} + m\frac{D-1}{2D} = \frac{N}{D} + m\frac{D-1}{2D}$$

(with N the total dynamic instruction count).

The second term in the above formula can be considered as an inherent dispatch inefficiency. The effect of this dispatch inefficiency on the average dispatch rate is illustrated in Figure 4.4, which shows the effective dispatch rate

$$D_{\mathit{eff}} = \frac{D}{1 + \frac{D-1}{2}\frac{m}{N}}.$$

The effective dispatch rate is smaller than the maximum dispatch rate of 4 because of the inherent dispatch inefficiency due to interval behavior; the effective dispatch rate decreases with an increasing number of miss events. The mcf benchmark for example has a large number of miss events (one miss event every 13 instructions on average), mainly due to long data cache misses, which results in an effective dispatch rate of 3.6; the effective dispatch rate for the other benchmarks, which have substantially fewer miss events, varies between 3.8 and 4.

4.2.2 Estimating miss event penalties

Computing the number of miss events is done by using specialized cache and branch predictor simulation. While some simulation is still needed, it can be done much faster than detailed simulation, since we only have to simulate one structure (in casu the branch predictor or the cache hierarchy) instead of the full processor (see Chapter 2). Furthermore, several predictor or cache sizes can be simulated simultaneously [92], and these simulations do not need to be redone when a processor parameter not related to caches and/or branch predictors (e.g., pipeline depth) changes.

The subsequent sections discuss the penalty estimation of the various miss event types in detail.

Instruction cache misses

The total penalty of instruction cache misses is the easiest to estimate. Interval analysis showed that the penalty of an instruction cache miss (or I-TLB miss) equals the miss delay. So, we just have to count the number of misses that are served by the L2 cache and those served by main memory, and multiply them by the access time of the L2 cache and main memory, respectively. The effect of the fetch buffer potentially offsetting one cycle of the penalty is considered a second-order effect and is not accounted for.

Interval analysis shows that we should not count instruction cache misses along mispredicted paths, but since a functional simulator does not simulate instructions along wrong paths anyway, instruction cache misses along these paths are not counted. Sometimes fetching instructions along mispredicted paths can prefetch instructions of the correct path (particularly in the case of reconvergent paths), removing (part of) the instruction cache miss penalty. This effect introduces a small error in the model (the difference in the number of instruction cache misses between functional and detailed simulation due to this effect incurs on average an extra estimation error of 0.2% on overall performance for the baseline configuration; the largest impact is observed for the benchmark gap, where the estimation accuracy improves by 1.54% if we eliminate the instruction cache misses prefetched by speculative execution).

Instruction cache misses can also be overlapped by long-latency loads, but this happens rarely, as shown in Section 3.5. In order to detect

such overlaps, instruction cache simulations should be done together with data cache simulations, which violates the idea of simulating each miss event in isolation, and improves accuracy only marginally (accuracy improves with 0.01% on average if we consider only the non-overlapped instruction cache misses compared to considering all instruction cache misses; the largest improvement due to this effect is 0.8%, and is again observed for gap).

Branch mispredictions

Branch misprediction penalties are a lot more difficult to estimate. As described in the previous chapter, they consist of the (constant) front-end pipeline depth and the (variable) branch resolution time. In order to estimate the branch misprediction penalty accurately, we have to estimate the branch resolution time for each branch misprediction individually.

The branch resolution time is the time between the dispatch of the branch and the execution of the branch, at which point in time the misprediction is detected. This is the time needed to execute all instructions the branch depends on, i.e., the dependency path leading to the mispredicted branch. Figure 4.5 shows the number of correct path instructions left in the issue buffer when a mispredicted branch is executed. Most of the time, this number is zero, indicating that the branch very often is the last instruction to execute. This means that the branch is on the critical dependency path of the instructions in the instruction window when the branch is dispatched. Therefore we can reuse the average critical path function $K(W)$ of Section 4.1, which provides the average critical path length for a given window size. Multiplying it by the average instruction latency ℓ gives the estimated resolution time. The only thing we need to know is the number of instructions residing in the instruction window when the branch is being dispatched.

To estimate the latter, we use the average critical dependency function, in the derived form of the attainable issue width as a function of the window size: $I(W) = \alpha/\ell \cdot W^{1-1/\beta}$. This formula gives the average number of instructions that can be issued when there are W instructions in the instruction window. To estimate the number of instructions in the window when the branch is being dispatched, we use the following iterative algorithm, which models the entering and exiting of instructions from the window:

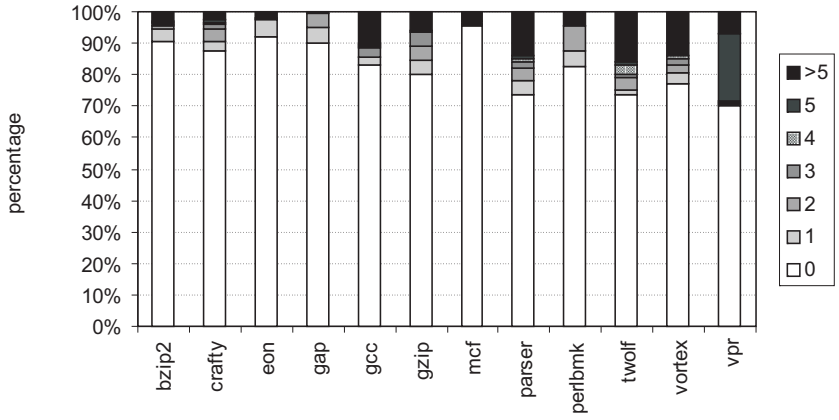

Figure 4.5: The number of correct path instructions left in the issue buffer when a mispredicted branch is executed.

1. The input of the algorithm is the $I(W)$ function, the interval length N (i.e., number of instructions between the previous miss and the branch misprediction), and the processor width D. In the beginning of the algorithm ($i = 0$), the number of instructions in the window W_0 is 0, and the number of instructions yet to be dispatched N_0 equals N. The total available size of the window is W instructions.

2. As long as $N_i \geq D$ and $W_i + D \leq W$, dispatch D instructions into the window: $W_{i+1} \leftarrow W_i + D$, and $N_{i+1} \leftarrow N_i - D$. If less instructions can be dispatched because there are not enough instructions left ($N_i < D$) or the window is almost full ($W_i + D > W$), dispatch the maximum possible number of instructions.

3. The number of instructions that can be issued and leave the window in the next cycle is on average $min(I(W_{i+1}), D)$ (the issue width is also bounded to D), so $W_{i+1} \leftarrow W_{i+1} - min(I(W_{i+1}), D)$.

4. If $N_{i+1} > 0$, increment i and go to step 2, else stop and output the window size W_{i+1}. The resolution time is estimated by $\ell K(W_{i+1})$.

This algorithm is like "filling a leaky bucket", every cycle instructions enter and leave the window. In the beginning, more instructions enter the window than there are instructions that can leave the window,

and the number of instructions in the window will grow. But as there are more and more instructions in the window, more independent instructions can be found, and the "leakage" will be bigger. At a certain point, the "leakage rate" is equal to the "filling rate" and steady state is reached, meaning that the number of instructions in the window remains roughly constant. If there are more than a certain amount of instructions in the interval, the number of instructions in the window will not grow anymore and the algorithm can be stopped prematurely. This can also happen if the window size is too small, such that it becomes full before steady state is reached. From then on, the window will always be full. This algorithm is similar to that used in [67, 68] to study instruction fetch requirements.

This algorithm provides for a given application (with average critical path parameters α and β, and average instruction latency ℓ) and a given window size W, the branch resolution time as a function of the interval length N. The average branch resolution time can then be calculated by measuring the branch misprediction interval length distribution of the program, and taking the average of the resolution times calculated by the algorithm, weighted by the occurrence probabilities of the interval lengths.

Since the distribution of the interval lengths is an input parameter to calculate the branch resolution time, we need to keep track of the interval lengths of branch misprediction intervals. This can be done during the simulation of the miss events. In the optimal case, interval lengths should be measured between the previous miss event (cache miss or branch misprediction) and the considered branch misprediction. We however decided to do the miss event simulations separately, so we measure the interval lengths since the previous branch misprediction. Like that, the branch predictor simulation results can be reused if a cache parameter is changed. Because branch mispredictions tend to occur in bursts, the extra error this incurs is very small (if we calculate the branch misprediction interval lengths considering all miss events, the average accuracy of the model improves with 0.05%; the largest accuracy improvement is 1.36%, again for gap).

Interval analysis and the algorithm above also provide interesting insights into the penalty of a branch misprediction. The following characteristics have an impact on the resolution time [32]:

1. *Interval length.* The more instructions dispatched since the previous miss and thus the longer the interval, the longer the depen-

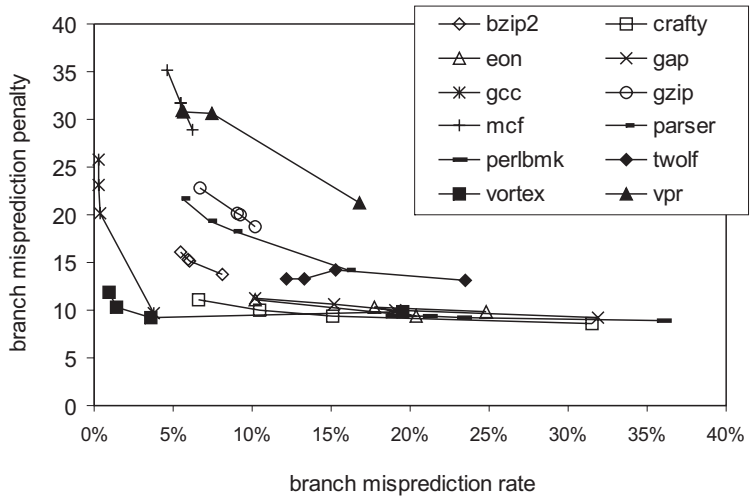

Figure 4.6: Average branch misprediction penalty as a function of branch misprediction rate.

dency path can grow. Interval length is in its turn determined by the number of miss events. Many miss events mean small intervals, whereas few miss events increase the number of instructions between the miss events. This means that the penalty for a branch misprediction becomes larger when there are fewer miss events. This is illustrated in Figure 4.6, showing the average branch misprediction penalty as a function of the branch misprediction rate. It shows how for most of the benchmarks the branch misprediction penalty decreases when the miss rate increases, especially in the 0%-10% misprediction rate range. A special case is twolf, where there is an increase in the penalty around a misprediction rate of 15%. This is because the branch misprediction penalty depends on the distribution of the interval lengths, and not only on the average interval length (due to the fact that the penalty is not linearly proportional to the interval length). Different branch predictors can cause a competely different interval length distribution, which means that an increased number of branch mispredictions does not always incur a smaller branch penalty.

2. *Instruction-level parallelism (ILP).* If a program has high ILP, the dependency paths are rather small and the resolution time is short. Programs with less ILP have longer dependency paths and there-

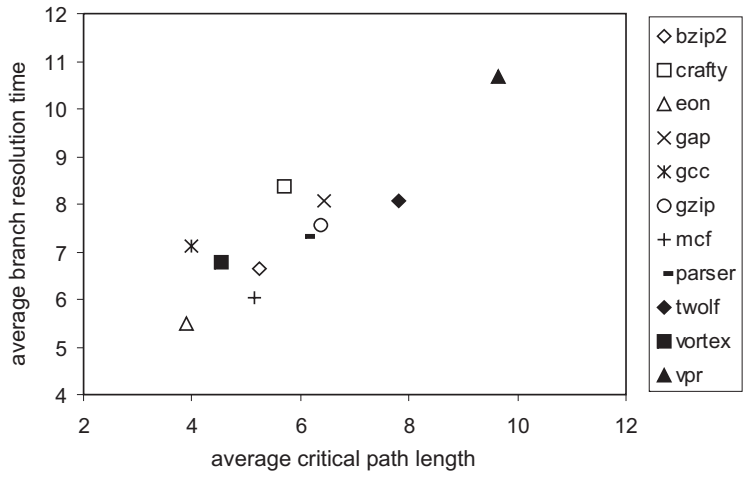

Figure 4.7: Average branch resolution time versus average critical path length.

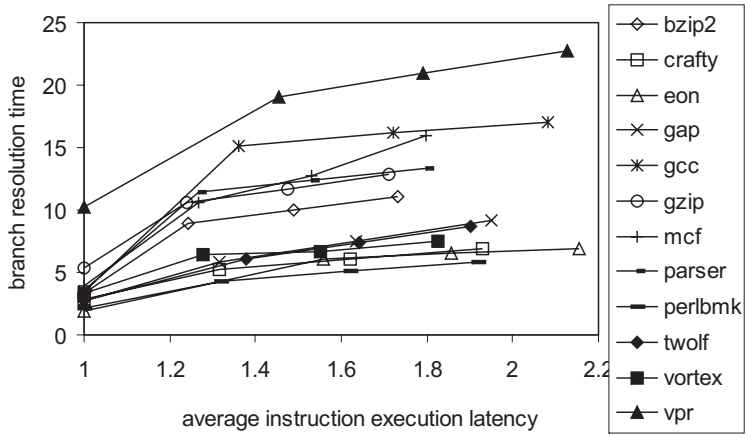

Figure 4.8: Average branch misprediction penalty as a function of average instruction latency.

fore larger branch misprediction penalties. This property is illustrated in Figure 4.7, which shows a strong correlation between the average critical path length and the branch resolution time (correlation coefficient of 0.85).

3. *Average instruction latency.* The execution time of a dependency path not only depends on its size, but also on the latencies of the

instructions that it consists of. The longer the average instruction latency of a program, the larger the branch misprediction penalty. This is illustrated in Figure 4.8. This figure is the result of a set of experiments where we first set the latency of each functional unit to 1 cycle—which means that the average instruction latency is one cycle—and we then gradually increased the latencies of the functional units (depending on its type). For all of these experiments, we calculated the average instruction latency as well as the average branch resolution time.

Long data cache misses

Interval analysis showed that the penalty of a long-latency load miss equals the resolution time of the load plus the memory access latency minus the time needed to fill the ROB, i.e., W/D (see Section 3.4.1). Since the resolution time and the ROB fill time are in the same range (a few tens of cycles), which is at least one order of magnitude smaller than the memory access latency, and they have opposite signs in the penalty calculation, we approximate the penalty of an isolated long-latency load miss by the main memory access time.

As discussed in the previous chapter, long-latency loads overlapping each other can have a large (positive) impact on performance. Therefore we should also estimate the amount of overlap (MLP). Assuming that the penalties of two or more independent long-latency load misses that can reside together in the ROB completely overlap, the MLP can be determined during cache simulation by stating that a long-latency load that occurs less than W instructions from another load miss and that is independent of that load miss, will be completely overlapped by the first load miss. As such we can compute the number of non-overlapped load misses. The total long-latency load penalty is then estimated as the number of non-overlapped misses multiplied by the main memory latency.

To determine the number of non-overlapped load misses efficiently, we developed the following algorithm that computes the number of non-overlapped misses for different ROB sizes (up to a maximum size) in one sweep. It uses a specialized functional simulator, that is able to detect register and memory dependencies and performs cache simulation to detect long-latency load misses.

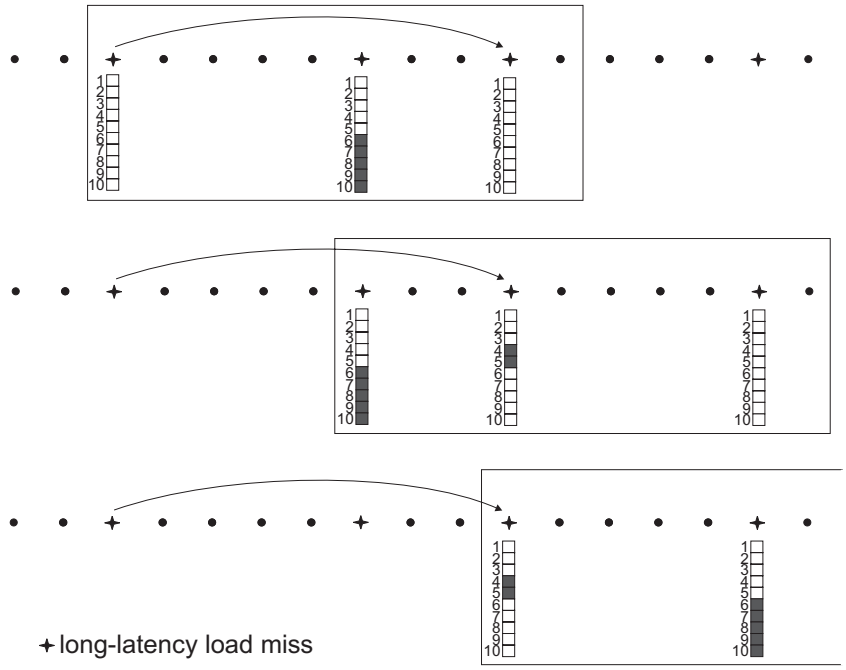

Figure 4.9: Example of MLP calculation algorithm. The maximum window size in this example is 10 entries.
Top part. The window shifts until the first long-latency load miss appears at the head. The first miss is never overlapped, as indicated by its bitmap (a white box means that this miss is not overlapped for that window size, a filled box means that it is overlapped for that window size). The second load miss will overlap with the first if the ROB size is equal to or larger than 6 entries, the bitmap is adapted to reflect this. The third load miss depends on the first miss and will therefore never overlap with the first miss.
Middle part. The window shifts until the next long-latency load miss. This load miss is overlapped if the ROB size is equal to or larger than 6, so we increment the number of non-overlapped misses for ROB sizes 1 to 5. This also means that it will only come at the head of the ROB if the ROB size is smaller than 6. This means that we should only look for overlaps within a window size of 5 entries. Since the next load miss does not depend on this load miss, it can overlap with the current load miss for ROB sizes 4 and 5. The last load miss will not overlap with the miss at the head of the window, since for this window size, the load miss at the head will be overlapped and will never hit the head of the ROB.
Bottom part. The window keeps on shifting. The load miss at the head of the window will be overlapped if the ROB size equals 4 or 5, so we increment the number of non-overlapped misses for ROB sizes 1-3 and 6-10. Since the last load miss is independent of the current load miss, it will be overlapped for ROB sizes from 6 to 10.

As in the critical path calculation algorithm, we start with considering the first window of maximum size, and determine all dependencies and long-latency loads. We then shift the window until the first long-latency load miss appears at the head of the window. All other long-latency load misses that occur within that window and that are independent of the miss at the head are marked as overlapped. To differentiate between different ROB sizes, we keep a bitmap per load miss of size equal to the maximum window size, that indicates for what window sizes this load miss is being overlapped. We refer to Figure 4.9 for an illustrative example of the calculation of these bitmaps.

Then we continue shifting the window until the next long-latency load miss is at the head. The corresponding bitmap of this load miss now indicates for what window sizes this load miss is not overlapped (those where the bits are zero), and the non-overlapped counters of that window sizes are incremented. Then again the window is searched for misses that overlap with the current load miss at the head of the window, and the bitmaps are adapted. This continues until the sliding window reaches the last instruction. The non-overlapped counters now contain the number of non-overlapped long-latency load misses per window size.

Short data cache misses

As explained in the previous chapter, short data cache misses incur no or a small penalty. Therefore we do not account for explicit penalties for short data cache misses in our model. However, they can have an impact on the branch misprediction penalty. When a branch misprediction depends on a short data cache load miss, the resolution time of the branch will be larger than if there was no miss. Therefore we also determine the number of short data cache misses (through specialized cache simulation) and account for them in the average instruction latency ℓ.

4.2.3 Overall model

The total execution time C, measured in cycles, can then be estimated as:

$$C = N/D + ((D-1)/2D)(m_{iL1} + m_{iL2} + m_{br} + m^*_{dL2}(W)) + \\ m_{iL1}\, c_{iL1} + m_{iL2}\, c_{L2} + \\ m_{br}\, (c_{br} + c_{fe}) + \\ m^*_{dL2}(W)\, c_{L2}. \tag{4.1}$$

with m_{iL1}, m_{iL2}, m_{br}, the number of L1 I-cache (IL1), L2 I-cache (IL2) and branch misses, and $m^*_{dL2}(W)$ the number of non-overlapped L2 D-cache (DL2) misses for window size W. c_{iL1}, c_{L2} is the latency of an IL1 miss and a L2 miss, c_{fe} is the front-end pipeline depth and c_{br} is the average branch resolution time. To keep the above formula simple, we did not include I-TLB and D-TLB misses, their contribution is calculated similarly to the penalties of the IL2 and DL2 misses.

Model inputs

The inputs to the model can be divided in three categories: application-related characteristics, processor configuration parameters and characteristics dependent on both the application and the processor configuration.

1. Application characteristics:
   - The number of instructions N.
   - The average critical dependency path $K(W)$, to calculate the branch resolution time c_{br}.
   - The instruction mix, to calculate the average instruction latency ℓ.
2. Processor parameters:
   - The dispatch width D.
   - The ROB size W.
   - The front-end pipeline depth c_{fe}.
   - The access time to the L2 cache c_{iL1} and main memory c_{L2}.
   - The execution unit latencies, to calculate the average instruction latency ℓ.
3. Application and processor dependent characteristics:

- The number of branch mispredictions m_{br} and the distribution of the number of instructions between the branch mispredictions (interval lengths) to calculate the branch resolution time.
- The number of IL1 (m_{iL1}), IL2 (m_{iL2}) and I-TLB misses.
- The number of DL1 misses, that together with the L2 access time contributes to the average instruction latency[2].
- The number of non-overlapped DL2 ($m^*_{dL2}(W)$) and D-TLB misses.

4.2.4 Model validation

Figure 4.10 compares the performance in IPC of different benchmarks on a 2-wide, 4-wide (baseline), 6-wide and 8-wide processor (see Appendix A) as obtained by detailed simulation with the results from the model described above. Figure 4.11 breaks down overall CPI in its (major) CPI components, for the baseline processor configuration. We observe that the interval model tracks the performance differences fairly well across benchmarks, processor configurations and CPI components.

The average IPC difference with respect to simulation for the baseline configuration is 6.9%. The largest differences are observed for parser (21.3%), gzip (16.9%), vpr (12.1%) and perlbmk (8.6%). For gzip and perlbmk, the cause of these differences is the difficulty in estimating the branch misprediction penalty, see Figure 4.11. This is due to the fact that the average critical dependency path and the average instruction latency do not represent well the specific dependency paths leading to mispredicted branches. For parser and vpr, the large error is mainly caused by an overestimation of the amount of overlapping long-latency loads. The cause is that for these benchmarks, the occurrence of dependent branch mispredictions and long instruction cache misses often stops the dispatching of (correct-path) instructions under the long-latency load miss, inhibiting the further exploitation of MLP. These effects are not taken into account in the MLP calculation algorithm (since they would require a simulation of all miss events together).

[2]The average instruction latency ℓ is then calculated as follows, with T the number of instruction / execution unit types, n_i the number of dynamic instructions of type i, lat_i the latency of functional unit i, m_{dL1} the number of DL1 misses and c_{dL1} the access time to the L2 cache: $\ell = \frac{1}{N}(\sum_{i=1}^{T} n_i\ lat_i + m_{dL1}\ c_{dL1})$.

The average IPC errors of the 2, 6 and 8-wide configurations are 3.7%, 9.4% and 12.5%, respectively. The estimation error increases with the width of the processor, this is mainly due to the fact that together with the width, we also sized up the ROB size in order to keep the processor configurations balanced (see Section 4.1 and Appendix A). This larger ROB size causes more overlap effects that are not modeled: there are more instructions dispatched under the long-latency load miss, i.e., the W/D term will be bigger than the load resolution time and has more impact, and more instruction cache misses and branch mispredictions may be overlapped by long-latency load misses (as indicated by Table 3.1). The MLP calculation mechanism also becomes less accurate as the window size grows, since more events that possibly inhibit the exploitation of MLP (dependent branch mispredictions, long instruction cache misses, etc.) may happen under the long-latency load miss, whereas the model assumes that the complete instruction window size can be used to exploit MLP.

4.3 Comparison to other models

A number of researchers have looked at superscalar processor models [20, 23, 43, 48, 52, 53, 56, 62, 67, 68, 73, 74, 87, 93], but three primary efforts led us to the performance model as described here. First, Michaud et al. [67, 68] focus on performance aspects of instruction delivery and modeled the instruction window and issue mechanisms. They show a power law (roughly a square law) relationship between the window size and the issue rate. This is in line with much earlier work dating back to [81] and [100]. Second, Karkhanis and Smith [56] extend this type of analysis to all types of miss events and build a complete performance model, which includes a sustained steady state performance rate punctuated with gaps that occur due to miss events. Third, Taha and Wills [93, 94] break instruction processing into intervals interrupted by branch mispredictions (which they call ‘macro blocks’), but they do not model the interval behavior of instruction and data cache misses.

The main difference with the model presented here is that they all study issue behavior to build a model. As we have shown in the previous chapter, dispatch behavior provides a sharper delimiter of intervals and requires no modeling of ramp-up and ramp-down issue behavior. In addition, the mechanistic model extends prior analytical su-

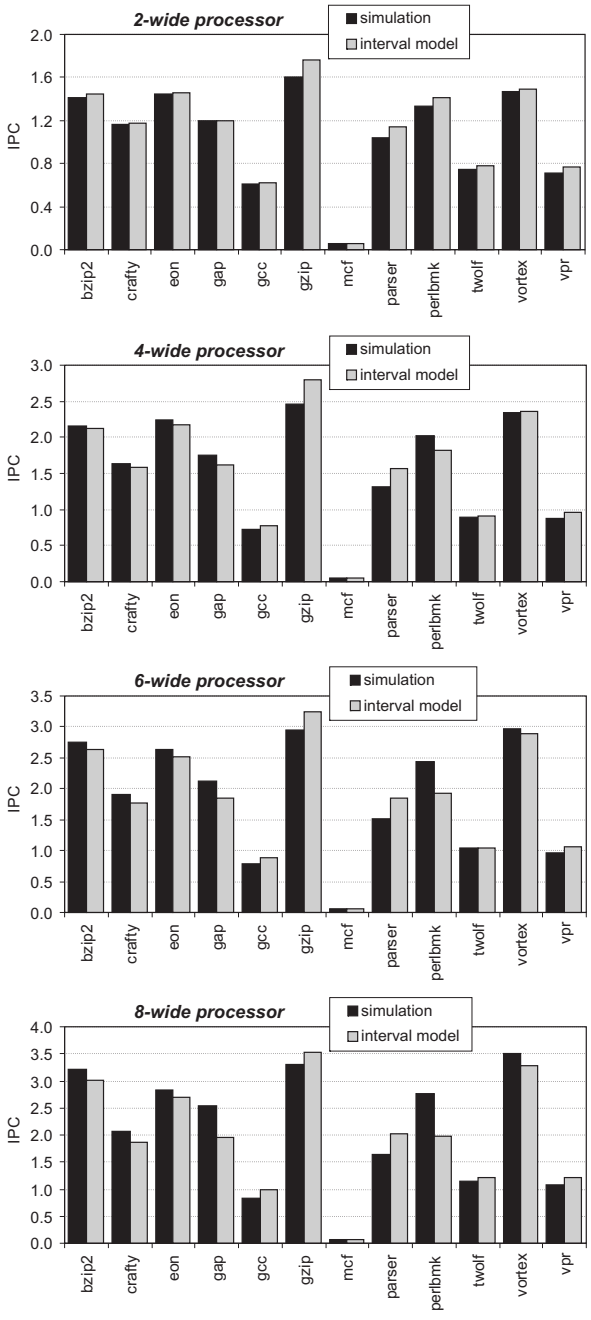

Figure 4.10: Comparing IPC predicted by the interval model versus simulation.

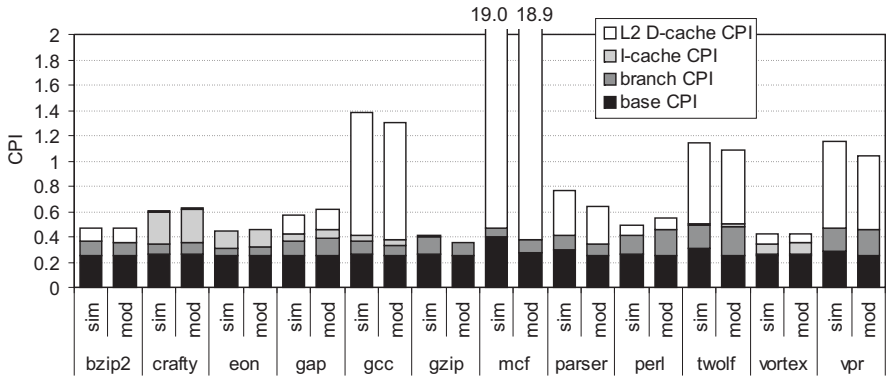

Figure 4.11: Comparing the CPI stacks predicted by the interval model ('mod') versus simulation ('sim') for the baseline 4-wide superscalar processor configuration.

perscalar processor modeling work in two major ways. First, interval analysis exposes the impact of the interval behavior on overall performance. This is reflected in how the interval model deals with (i) the dispatch inefficiency—more miss events and thus more intervals increase the dispatch inefficiency—and (ii) the branch resolution time—longer interval lengths result in longer branch resolution times. Second, our mechanistic model estimates the branch resolution time using the average critical path length $K(W)$ that characterizes a program's inherent ILP. No extensive detailed processor simulations are needed as done in [56] and [94] for computing the IW characteristic.

Karkhanis and Smith [55] use the performance model described in this chapter to study the automatic design of application-specific superscalar processors, in combination with an energy estimation model. They show that the performance model can be used to perform fast design space exploration, and that it is able to mark the Pareto-optimal design points.

Another, more common, way of modeling superscalar processor performance is by using empirical models [43, 48, 62]. These models are based on parameterized equations or general functions, which are trained using performance simulations of a number of processor configurations. The fit equations or functions can then be used to estimate the performance of other processor configurations. The model can be based on simple intuitive equations [43], or use more general automatically deduced equations or functions (e.g., regression theory [62] or

neural networks [48]). The main advantage of this approach is that the model is automatically generated, it does not need a detailed performance model. The disadvantages of such models compared to a mechanistic model are that (i) still a number of detailed simulations are needed to train the model (which takes considerably more time than the functional simulations to locate the miss events, needed for the mechanistic model; this also means that a detailed simulator has to exist or has to be developed in order to use these models, the mechanistic model only needs simpler functional simulators), (ii) the automatically deduced equations provide little insight into the underlying mechanisms that determine performance, and (iii) it gives no insight for which processor configurations the model is valid.

Although developed independently, the equations in the mechanistic performance model have a similar structure to those in the empirical model of Hartstein and Puzak [43]. Because the construction of the Hartstein-Puzak model has intuitive appeal, we intentionally organize the components of the interval model along similar lines in the next section, where we will also compare this model to our model.

4.4 Application: pipeline depth and width study

We now use the mechanistic performance model to study the impact of pipeline depth and width. In this section, we assume balanced processor designs, which means that we should scale up the processor resources as we deepen and/or widen the processor pipeline, see Section 4.1. Appendix A shows how we scaled up different resources as the processor pipeline becomes deeper and wider.

4.4.1 Pipeline depth

Although the optimum pipeline depth is a well-researched topic [21, 26, 44, 88, 90], the mechanistic model provides additional insights into how pipelining affects individual microarchitecture components, and in addition, provides the opportunity to study partial pipelining, which other models do not include.

We first extend the mechanistic model by converting from clock cycles to absolute time (say, in nanoseconds). To do so, we multiply the estimated cycle counts by the absolute cycle time. We denote the cycle time as $t_S = t_o + t_p/p$, with t_o the latch overhead for a given tech-

nology, t_p the total logic delay of the processor, and p the number of pipeline stages. As such, the total execution time then becomes, organizing the terms along the lines of the model proposed by Hartstein and Puzak [43]:

$$\begin{aligned} T = [N/D + (D-1)/2D)(m_{iL1} + m_{iL2} + m_{br} + m^*_{dL2}(W)](t_o + t_p/p) + \\ m_{iL1}\, c_{iL1}\, p/p_b\, (f_{iL1}\, t_o + t_p/p) + m_{iL2}\, c_{L2}\, p/p_b\, (f_{iL2}\, t_o + t_p/p) + \\ m_{br}\, (c_{br}(p,D) + c_{fe})\, p/p_b\, (t_o + t_p/p) + \\ m^*_{dL2}(W)\, c_{L2}\, p/p_b\, (f_{dL2}\, t_o + t_p/p). \end{aligned} \tag{4.2}$$

In this equation, p_b is the number of pipeline stages in the baseline configuration; the same holds for c_{iL1}, c_{L2} and c_{fe}. The pipeline factors f ($0 \leq f \leq 1$) denote the degree to which the cache misses are pipelined; $f = 0$ denotes non-pipelined cache misses and $f = 1$ denotes fully pipelined cache misses. If all units are always fully pipelined (as is sometimes assumed), then the f terms can be removed to simplify the equations. Overall TPI (time per instruction) is then computed by dividing the above total execution time T by the total dynamic instruction count N: $TPI = T/N$.

We now discuss how the four terms in the above equation are affected by pipeline depth for balanced processor designs. We also refer to Figure 4.12 which shows how the most various TPI components are affected by pipeline depth, averaged across the SPEC CPU2000 integer benchmarks. The top graph in Figure 4.12 assumes pipelined cache misses, whereas the bottom graph assumes non-pipelined cache misses. The indicated number of pipeline stages reflects the number of front-end pipeline stages. To further demonstrate the accuracy of the model as a function of pipeline depth, we also show the overall TPI numbers obtained through simulation in the top graph.

Base TPI. The first term (base TPI) is where the performance advantage of pipelining appears. A higher clock frequency reduces the absolute time for performing useful work.

I-cache TPI. The I-cache TPI terms remain constant in terms of cycles (the p in the numerators and denominators cancel) but increase linearly in actual time because of pipelining overhead, in case of (partially or fully) pipelined cache misses, see top graph in Figure 4.12. In case of non-pipelined cache misses, the I-cache terms remain constant in absolute time, see bottom graph in Figure 4.12.

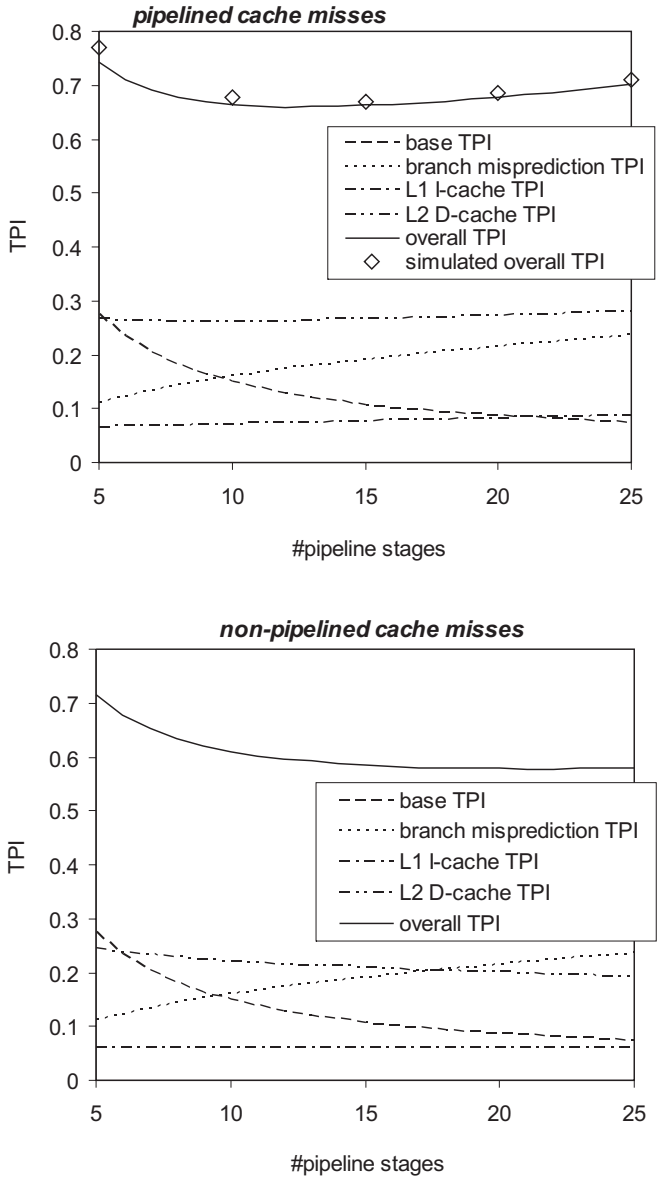

Figure 4.12: TPI components and overall TPI as a function of depth, assuming pipelined cache misses (top graph) and non-pipelined cache misses (bottom graph).

Branch misprediction TPI. The branch misprediction term can be broken up into two terms, one related to window drain (branch resolution time) and another one related to refilling the front-end pipeline. The pipeline refill term, $m_{br}\ c_{fe}\ p/p_b\ (t_o + t_p/p)$, increases with increasing pipeline depth because of pipeline overhead. The window drain term, $m_{br}\ c_{br}(p, D)\ p/p_b\ (t_o + t_p/p)$, also increases with pipeline depth for two reasons. First, in a balanced processor, the ROB accumulates more instructions when increasing pipeline depth, and by consequence, draining more instructions takes longer. Second, draining instructions with a deeper pipeline also increases drain time (counted in cycles). In other words, and in contrast to conventional wisdom, the impact of pipeline depth on the branch misprediction penalty is variable and is larger than its effect on the pipeline refill term; the branch resolution time also increases with deeper pipelines. Figure 4.12 in its top graph further illustrates this finding: if the pipeline refill term would be the only contributing term, then the slopes for the branch misprediction TPI and the I-cache TPI would be identical; however, the slope for the branch misprediction TPI is much steeper than the slope for the I-cache TPI.

Long-latency back-end TPI. The long back-end D-cache TPI term is affected by pipeline depth in two major ways. First, the cache miss latency term may be partially pipelined or non-pipelined. Like the I-cache TPI terms, if (partially) pipelined, this term increases in actual time because of pipelining overhead; if non-pipelined, this term remains constant irrespective of pipeline depth. Second, the number of non-overlapping long load misses m^*_{dL2} may decrease with deeper pipelines because of the larger ROB sizes for balanced processor designs. In other words, more memory-level parallelism (MLP) gets exposed. Figure 4.12 illustrates this very well. The bottom graph assuming non-pipelined cache misses shows that the long-latency load TPI decreases with pipeline depth because of increased MLP. The top graph assuming pipelined cache misses shows that the increased MLP is offset by the increased pipelining overhead: initially the long-latency TPI decreases and then increases with increased pipeline depth.

Pipelined versus non-pipelined cache misses. Another observation from Figure 4.12 is that the optimum pipeline depth assuming non-pipelined cache misses is larger than assuming pipelined cache misses. The reason is that the I-cache miss TPI remains constant and the long-latency load TPI decreases with deeper pipelines for non-pipelined cache misses which makes the optimum pipeline depth shift towards deeper pipelines.

Comparison to prior work on pipeline depth.

Prior work on pipeline depth uses simulation [1, 46, 60, 88], or modeling [21, 26, 43, 90]. Early studies [21, 26, 60] investigated the impact of instruction dependencies, branching and pipelining overhead on optimal pipeline depth (for scalar and vector in-order processors). They found the optimal pipeline stage depth is about 8 gate levels [60]. This result was also found by more recent studies [1, 46, 88] on superscalar out-of-order processors, which also study the impact of branch mispredictions and cache misses (based on simulation).

The pipeline depth model closest to our model presented in this study is the model by Hartstein and Puzak [43]. Hartstein and Puzak divide the total execution time in busy time T_{BZ} and non-busy time T_{NBZ}. The busy time refers to the time that the processor is doing useful work, i.e., instructions are issued; the non-busy time refers to the time that execution is stalled by miss events (called hazards by Hartstein and Puzak). Hartstein and Puzak derive that the total execution time equals busy time plus non-busy time:

$$\begin{aligned} T = N_{total}/\alpha\ (t_o + t_p/p) + \\ N_H\ \gamma\ (t_o p + t_p) \end{aligned} \tag{4.3}$$

with N_H the number of miss events; α and γ are empirically derived by fitting the model versus simulation. Comparing Formula 4.3 against Formula 4.2, we observe that the mechanistic interval model provides more insight than the empirical model by Hartstein and Puzak. For example, according to Hartstein and Puzak, the parameter α is not the superscalar issue width, but the 'actual degree of superscalar processing'. The interval model shows that α in fact is the achieved dispatch efficiency which is a function of processor width and the number of intervals. Similarly, Hartstein and Puzak denote the parameter γ as the fraction of the total pipeline delay stalled due to a miss event averaged across all miss events. In other words, Hartstein and Puzak assume that all miss event types affect performance the same way. The mechanistic model presented in this chapter on the other hand, factors out the different types of miss events. An important consequence of separating miss event effects in the mechanistic model, is that it enables modeling partial pipelining of cache misses which is impossible using the empirical Hartstein and Puzak model.

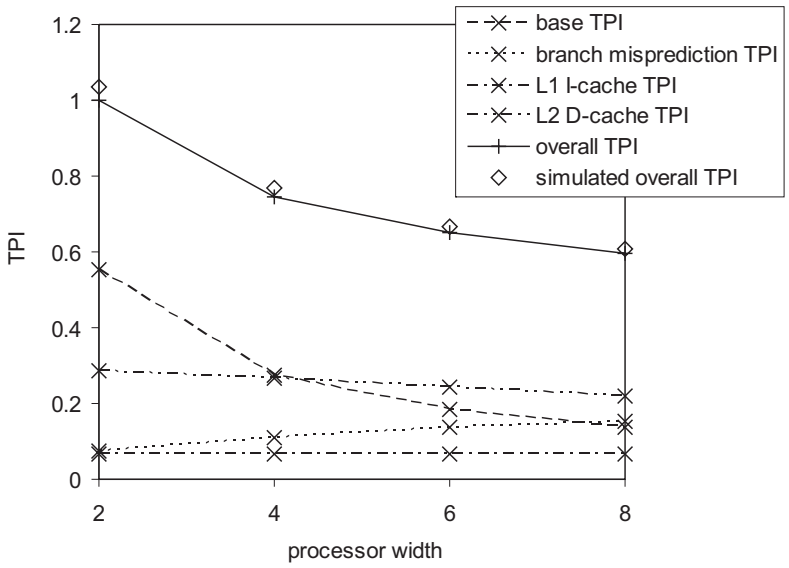

Figure 4.13: TPI components and overall TPI as a function of width.

4.4.2 Pipeline width

We now discuss how the various TPI components are affected by width. The main performance advantage of increasing width D of superscalar processing comes from the first term of Formula 4.2—doubling the processor width halves the execution time for performing useful work. This is also illustrated in Figure 4.13 which shows how the various TPI components are affected by processor width, averaged across all benchmarks. The second term that decreases with increasing width is the long-latency load miss term. The number of non-overlapping long back-end misses decreases with increasing widths, the reason being the increased ROB size for balanced processor designs.

There are two terms that increase with increasing processor width: the dispatch inefficiency term and the branch misprediction term. The most significant term is the branch misprediction term which increases because of an increased window drain time. Recall that for a balanced processor design, the ROB accumulates more instructions with an increased processor width and draining more instructions from the ROB takes longer—the increased processor width does not help the drain time because the ROB cannot be drained faster than the instructions'

critical path length.

4.4.3 Optimal pipeline depth/width for balanced processors

Having discussed the effect of pipeline depth and width on overall performance separately, we now discuss the interaction between pipeline depth and width. Figure 4.14 shows overall TPI as a function of pipeline depth for different processor widths. These graphs clearly show that the optimum pipeline depth shifts towards shallower pipelines for increasing processor widths. The reason for the phenomenon is that the base TPI component decreases with increasing width, while the miss event penalty components do not. This inverse relationship between processor width and pipeline depth for optimal performance can be theoretically derived by differentiating Formula 4.2 to p. Setting this equal to zero, and solving this equation to p yields the optimal pipeline depth p^*. This results in the following equation, assuming fully pipelined cache misses:

$$p^* = \sqrt{\frac{t_p t_b}{t_o} \frac{\frac{N_{total}}{D} + \frac{D-1}{2D}(m_{iL1} + m_{iL2} + m_{br} + m^*_{dL2}(W))}{m_{iL1}\, c_{iL1} + m_{iL2}\, c_{L2} + m_{br}\, (c_{dr} + c_{fe}) + m^*_{dL2}(W)\, c_{L2}}}; \quad (4.4)$$

or, assuming non-pipelined cache misses:

$$p^* = \sqrt{\frac{t_p t_b}{t_o} \frac{\frac{N_{total}}{D} + \frac{D-1}{2D}(m_{iL1} + m_{iL2} + m_{br} + m^*_{dL2}(W))}{m_{br}\, (c_{dr} + c_{fe})}}. \quad (4.5)$$

After eliminating the insignificant and irrelevant terms, the optimal pipeline depth for a given processor width p^* is approximately proportional to the processor width D to the power $-1/2$, i.e., $p^*(D)\sqrt{D} \simeq constant$. Note that this relationship holds for both pipelined as well as non-pipelined cache misses. In other words, when increasing the processor width by a factor c, one must decrease the pipeline depth by a factor $\sqrt{c}$ in order to achieve the optimum depth for the given width. Figure 4.14 validates this finding. For example, for the pipelined cache miss case, the optimal pipeline depth for the 8-wide machine (9 stages) is approximately a factor $\sqrt{2}$ smaller than the optimal pipeline depth for the 4-wide machine (12 stages); and, the optimal depth for the 4-wide machine is approximately a factor $\sqrt{2}$ smaller than the optimal pipeline depth for the 2-wide machine (17 stages).

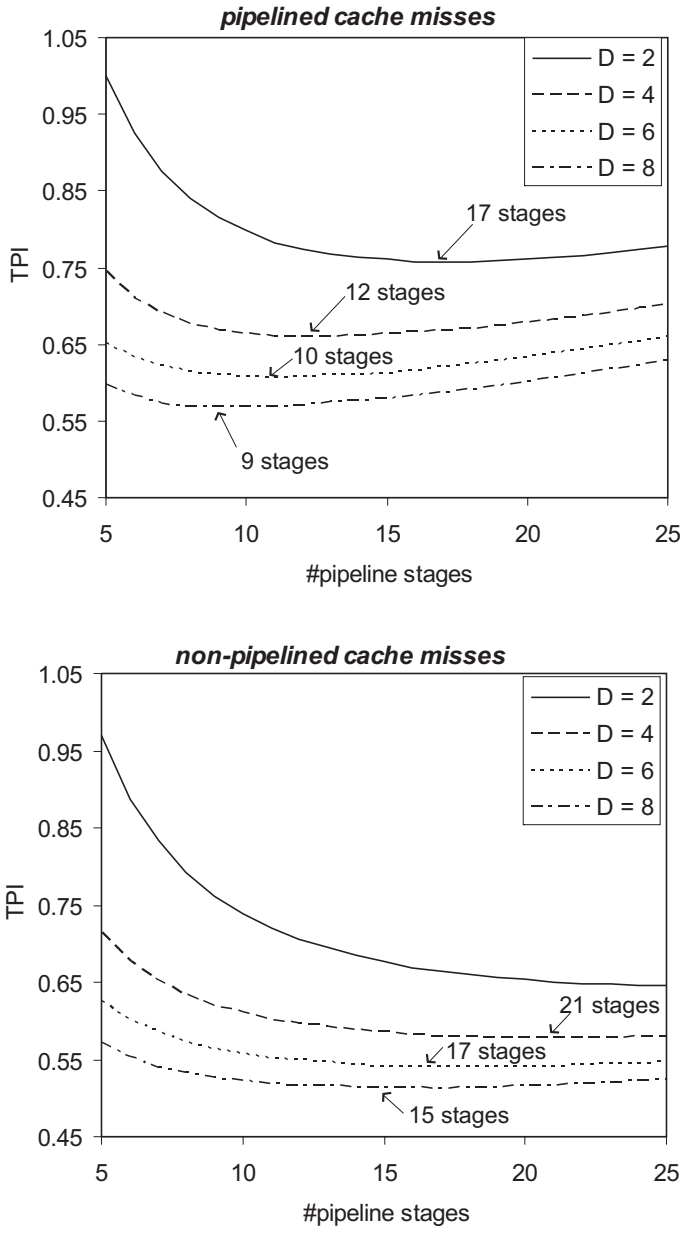

Figure 4.14: Overall TPI as a function of depth and width, assuming pipelined cache misses (top graph) and non-pipelined cache misses (bottom graph).

4.5 Summary

In this chapter, we developed a performance model based on the insights provided by interval analysis. It estimates the performance of an application running on a superscalar processor. It does so by first counting the number of miss events (instruction cache misses, data cache misses and branch mispredictions) through specialized functional simulations. Then the penalties of the individual miss events are estimated as follows:

- For *instruction cache misses*, the penalty equals the access time to the lower level cache or main memory.
- The penalty of a *branch misprediction* consists of the front-end refill time and the branch resolution time. The former is a configuration parameter of the processor and is fixed. The latter is variable and depends on application characteristics and the processor configuration. We approximate the branch resolution time as the window drain time, because when a branch is mispredicted, that branch is often the last instruction to be executed. We developed an algorithm to estimate the window drain time using the interval size distribution, the average critical path length distribution and the average instruction latency.
- The *long-latency load miss* penalty is approximated by the main memory access time.
- We also take into account overlaps between long-latency load misses, by calculating the number of independent long-latency load misses that can be in-flight concurrently within the reorder buffer.
- Since short data cache misses and medium-latency instructions incur no or small penalties, we do not account for them in the total execution time estimation. They are however important in the calculation of the branch resolution time.
- We ignore overlaps between front-end and back-end misses, since they are relatively infrequent.

In addition to the miss event penalties, we also need to estimate the base cycles, i.e., the number of cycles where instructions are dispatched. This is done by dividing the total number of instructions by

the dispatch width. We also compensate for the inherent dispatch inefficiency due the fact that interval sizes are not always a multiple of the dispatch width.

The model can accurately estimate the performance of a range of processor configurations (an average estimation error of 6.9% for the baseline 4-wide superscalar out-of-order processor). It also gives insight into the impact the various factors have on performance. We also showed how it can be used in the processor design process to determine the optimal pipeline depth and width, both for pipelined and non-pipelined cache misses. From the equations of the model, we also derived the insight that the optimal pipeline depth is inversely proportional to the square root of the processor width.

Chapter 5

Cycle accounting in single-threaded processors

Think? Why think?
We have computers to do that for us.
Jean Rostand

Estimating performance is just one application of interval analysis. As stated before, interval analysis also provides more insight into how different miss events affect performance. In particular, it quantifies the exact cycle penalty of each miss event. This can be used to do cycle accounting: of all cycles a program needed to execute, how many cycles have gone to doing actual work and how many cycles have been lost due to the various miss events?

This chapter proposes a hardware performance counter architecture to construct accurate cycle component stacks, driven by the insights from interval analysis. These stacks break up the total number of cycles into different cycle components, each reflecting the performance impact of the various components that determine overall performance, i.e., the cycles needed to execute the instructions (in the absence of miss events) and the cycles needed to handle the different miss events. A cycle component stack is a very intuitive way to visualize and analyze processor performance [6, 25, 27] (often referred to as a "CPI stack" and its components "CPI adders").

Figure 5.1 shows an example of a cycle component stack for the twolf benchmark. It reveals that of the 135 million cycles it needed to execute, 30 million were needed to execute the instructions in the absence

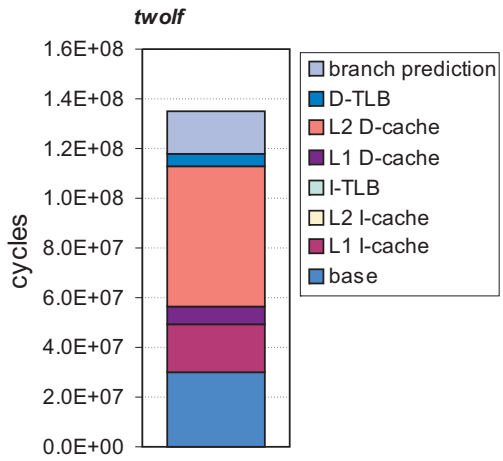

Figure 5.1: Example cycle component stack for the twolf benchmark.

of miss events (base component), and that the miss events that had the greatest impact on performance were L2 D-cache misses (56 million cycles), L1 I-cache misses (19 million cycles) and branch mispredictions (17 million cycles).

Cycle component stacks can be very helpful to software and compiler builders to analyze overall performance and detect the performance bottlenecks in the program. It can for example point out which code optimizations can have the greatest impact (e.g., for the twolf example, the performance can be improved by at most 41% through the reduction of the number of L2 D-cache misses). An accurate performance analysis at runtime is also useful for processor designers. If the performance counter architecture is built into the first produced hardware prototype, it can be used to validate assumptions made at the early stage architectural design process, and to perform more experiments on the new hardware, for which there was no time during architectural design (because in that design stage, we only have simulators to evaluate performance, which are very time-consuming).

Cycle accounting on a superscalar out-of-order processor, however, is more difficult than on an in-order processor, since there are more possible overlap effects between instruction execution and miss events, and between miss events mutually. Interval analysis enables us to isolate and quantify the performance impact of the various miss events. Existing hardware performance counter architectures however are un-

able to measure the correct values needed for the model, so a new hardware performance counter architecture has to be designed [29, 30]. The new cycle accounting architecture will be described in Section 5.1. Previously proposed cycle accounting methods will be discussed in Section 5.2, and the new performance counter architecture will be validated and compared to these other cycle accounting methods in Section 5.3. Related work is discussed in Section 5.4. The last section of this chapter illustrates the use of this counter architecture to study the impact of compiler optimizations on performance.

5.1 Performance counter architecture for computing accurate cycle components

To compute accurate cycle components, we need a new performance counter architecture that is able to accurately measure the penalties of the individual miss events. For each type of miss event there is a global counter that accumulates the cycle penalties of that type of miss event. There are 8 global counters, for counting the lost cycles due to L1 instruction cache misses, L2 instruction cache misses, I-TLB misses, L1 data cache misses, L2 data cache misses, D-TLB misses, branch mispredictions and other resource stalls. As discussed previously, there are two main categories of miss events that each need a different approach: front-end and back-end miss events.

5.1.1 Front-end miss events

Front-end miss events consist of instruction cache misses and branch mispredictions. The penalty of an instruction cache miss equals the number of cycles that is needed to fetch the instruction from the next level in the memory hierarchy. The only thing to be aware of is not to count instruction cache misses along mispredicted paths, because they do not contribute to the total instruction cache miss penalty (see Section 3.5.2). To account for that, we propose the *front-end miss event table (FMT)*, see Figure 5.2, which contains the number of cycles fetch was stalled because of instruction cache misses per in-flight basic block.

Each entry of the FMT is identified by the ROB identifier of the branch that ends the corresponding basic block, and contains counters to count IL1, IL2 and I-TLB misses. Three pointers are needed: a fetch tail pointer, a dispatch tail pointer and a dispatch head pointer. Initially

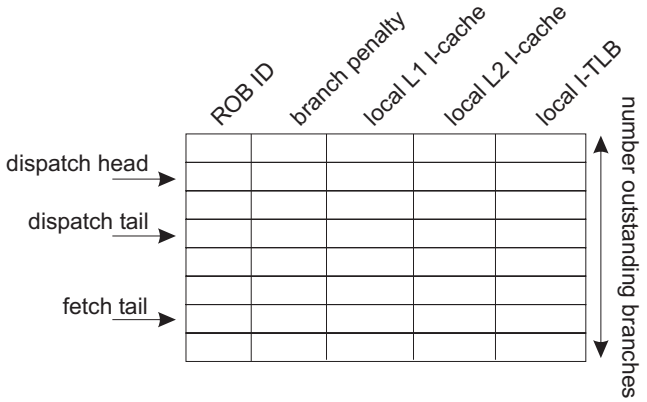

Figure 5.2: Front-end miss event table FMT.

these three pointers all point to the first entry. Each cycle the fetch stage is stalled due to an instruction cache miss, the corresponding counter pointed to by the fetch tail pointer is incremented. By doing so, the miss delay computed in the local counters corresponds to the actual instruction cache miss penalty.

The fetching of a branch indicates the end of the current basic block. Therefore the fetch tail pointer is incremented each time a branch is fetched. When the branch dispatches, we know the identifier of the ROB entry that contains the branch, and that identifier is written into the ROB ID field of the entry pointed to by the dispatch tail pointer. Then the dispatch tail pointer is also incremented. The entries between the fetch and dispatch tail pointer thus correspond to basic blocks that are fetched but not yet dispatched.

When a branch commits, it was not on a mispredicted path and so are all instructions in the basic block preceding this branch. In that case, the corresponding local instruction cache and I-TLB counters can be added to the global counters. The dispatch head counter is incremented, deallocating the committed entry. Upon the detection of a branch misprediction, the corresponding entry is looked up via the ROB ID. All subsequent entries are reset and the fetch and dispatch tail pointers are placed at the entry following the mispredicted branch. By doing so, instruction cache misses along mispredicted paths are not counted.

Branch misprediction penalties are also measured using the FMT. An extra ‘branch penalty’ field per entry counts the number of cycles

since the corresponding branch is dispatched. This can be done by incrementing the branch penalty field of each entry between the dispatch head and tail pointer every cycle. If the branch turns out to be correctly predicted, this value is not needed, but if it is mispredicted, it contains the number of cycles between the dispatch of the branch and the detection of the misprediction, i.e., the branch resolution time. In that case, the branch penalty counter is added to the global counter. From then on, this global counter is incremented every cycle until new instructions enter the ROB; this accounts for the front-end refill time.

Improved design

The FMT design needs a few hundred bits to store all local counters for every in-flight basic block. This can be reduced by using one local counter per instruction cache miss event in addition to the global counters, instead of local counters per FMT entry. This design, called the *shared FMT* or sFMT (see Figure 5.3), operates in a similar way as the FMT: upon an instruction cache miss, the cycles are counted in the local counter. We also keep track of the instructions that caused an instruction cache miss by providing an I-cache miss bit in every ROB entry (as done in the Intel Pentium 4 and IBM POWER5 for tracking I-cache misses in the commit stage). When an instruction with that bit set commits, we add the local counters to the global counters. The local counters are reset, and so are the I-cache miss bits of all instructions in the ROB. This has to be done because the local counters contain the total instruction cache miss penalties for all in-flight instructions. When a branch misprediction is detected, the branch misprediction penalty is calculated using the local branch penalty counter, as in the FMT case, and the local instruction cache miss counters are reset to avoid counting instruction cache misses along mispredicted paths.

This design can both overestimate and underestimate the instruction cache miss penalty. When an instruction cache miss is followed by a mispredicted branch that in its turn is followed by another (wrong-path) instruction cache miss, then there are two possibilities: (i) the first instruction cache miss commits before the branch misprediction is detected, or (ii) the misprediction is detected before the instruction cache commits. In the first case, the penalty of the second wrong-path instruction cache miss will be added to the global counters. In the second case, the local counters will be reset, and the penalty of the first cache miss is not counted. This is however an infrequent situation, given that

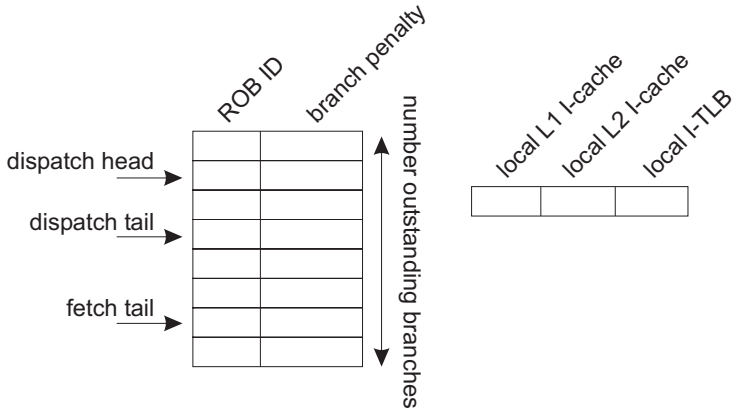

Figure 5.3: Shared front-end miss event table sFMT.

instruction cache misses and branch mispredictions typically occur in bursts.

The total additional error the sFMT introduces will be quantified in the validation section.

5.1.2 Back-end miss events

The penalty of a back-end miss event is equal to the time dispatch is stalled because the ROB is full (see Section 3.4.1 on page 32). This can be fairly simply implemented in the proposed hardware performance counter architecture: increment the respective global counter each cycle the ROB is full. The type of miss event (L1 D-cache, L2 D-cache or D-TLB miss, or other resource stall (e.g., medium-latency functional unit)) that caused the ROB to fill up, can be found by looking at the instruction blocking the head of the ROB. If it is waiting for data returning from a lower level cache or memory, the corresponding cache miss counter is incremented, and otherwise the cycle is counted as a resource stall. Note that this naturally handles overlapping long-latency misses: only the penalty of the first miss is counted, since overlapped misses do not block the head of the ROB.

5.1.3 Overlaps between miss events

One thing we should be aware of is front-end misses overlapped by back-end misses and vice-versa (see Section 3.5.2 on page 40). If we

would account cycles for both overlapping miss events, we would double-count these cycles and the sum of all cycle components would be larger than the total cycle count. Therefore we should attribute these cycles to only one miss event. Which miss event to choose is of less importance, since Table 3.1 illustrated the relative infrequency of these overlaps.

We first consider instruction cache misses that are overlapped by long-latency load misses. An instruction cache miss can only occur when instructions can still be fetched, which means the ROB is not full, in which case we do not account cycles to long-latency load misses. So we never account a cycle to both instruction cache misses and long data cache misses. Using this default behavior, we account the cycles where the instruction cache miss is handled concurrently with the long data cache miss to the instruction cache miss component. The remaining penalty cycles, i.e., when dispatch is stalled due to the full ROB, are then accounted to the long-latency load miss component.

If a long-latency back-end miss is followed by a branch misprediction before the ROB is full, then there is a possibility we count some cycles twice: if, after dispatching some wrong-path instructions, the ROB gets full, we count the following cycles as a back-end miss cycles, and, in addition, each cycle we increment the branch penalty counter of the local FMT entry, which will eventually be added to the global counter. This means that we account these cycles both to the long-latency load miss and the branch misprediction. Therefore we choose not to increment the branch penalty counter when the ROB is full. This means that we account cycles to the branch misprediction as long as the ROB is not full. If the branch misprediction was independent of the load miss (see Figure 3.17), the number of cycles where the ROB is full before the branch misprediction is detected will be zero or limited (because the branch miss will be detected long before the load miss returns). If the branch misprediction depends on the load miss (see Figure 3.18), then this scheme accounts the penalty exactly as indicated by the figure, i.e., the cycles where wrong-path instructions are dispatched are accounted to the branch misprediction, and the cycles the ROB is full are accounted to the long-latency load miss.

5.2 Other cycle accounting methods

The question "Where have all the cycles gone?" is not new and many researchers have tried to construct a cycle accounting method to accurately assign cycle penalties to miss events. A widely used naive approach for computing the various components is to multiply the number of miss events of a given type by an average penalty per miss event [2, 57, 64, 80, 103]. For example, the L2 data cache miss cycle component is computed by multiplying the number of L2 misses with the average memory access latency; the branch misprediction contribution is computed by multiplying the number of branch mispredictions with an average branch misprediction penalty. We will refer to this approach as the *naive approach* in the validation section.

There are a number of pitfalls to the naive approach, however. First, the average penalty for a given miss event may vary across programs, and, in addition, the number of penalty cycles may not be obvious. For example, interval analysis showed that the branch misprediction penalty can be substantially larger than the front-end pipeline length—taking the front-end pipeline length as an estimate for the branch misprediction penalty leads to a significant underestimation of the real branch misprediction penalty. Second, the naive approach does not consider that some of the miss event penalties can be hidden through out-of-order processing of independent instructions and miss events. For example, as illustrated in Chapter 3, L1 data cache misses can be hidden almost completely in a balanced, out-of-order superscalar processor. Also, two or more long-latency loads may overlap with each other. Not taking these overlapping miss events into account can give highly skewed estimates of the cycle components. And finally, the naive approach makes no distinction between miss events along mispredicted control flow paths and miss events along correct control flow paths. A naive method may count events on both paths, leading to inaccuracy.

To overcome the latter problem, some processors, such as the Intel Pentium 4 [89], feature a mechanism for obtaining non-speculative event counts. This is achieved by implementing a tagging mechanism that tags instructions as they flow through the pipeline, and the event counters get updated only when the instruction reaches the commit stage. If the instruction is not committed, i.e., the instruction is from a misspeculated path, the event counter does not get updated. We will refer to this approach as the *naive_non_spec* approach; this approach dif-

fers from the naive approach in that it does not count miss events along mispredicted paths.

In response to some of the above shortcomings, the designers of the IBM POWER5 microprocessor implemented a dedicated hardware performance counter mechanism with the goal of computing cycle component stacks [65, 66]. To the best of our knowledge, the IBM POWER5 is the only out-of-order processor implementing a dedicated hardware performance counter architecture for measuring cycle components. The IBM POWER5 has hardware performance counters that can be programmed to count particular commit stall conditions such as I-cache miss, branch misprediction, L2 D-cache miss, L1 D-cache miss, etc. The general philosophy for the IBM POWER5 cycle accounting mechanism is to inspect the commit stage of the pipeline, and if no instructions can be committed in a given cycle, the appropriate commit stall counter is incremented. As such, the commit stall counters count the number of stall cycles for a given stall condition. There are two primary conditions for a commit stall.

- First, the reorder buffer is empty. There are two possible causes for this.
  - An I-cache miss, or an I-TLB miss occurred, and the pipeline stops feeding new instructions into the ROB. This causes the ROB to drain, and, eventually, the ROB may become empty. When the ROB is empty, the POWER5 mechanism starts counting lost cycles in the I-cache completion stall counter until instructions start entering the ROB again.
  - A branch is mispredicted. When the mispredicted branch gets resolved, the pipeline needs to be flushed and new instructions need to be fetched from the correct control flow path. At that point in time, the ROB is empty until newly fetched instructions have traversed the front-end pipeline to reach the ROB. The POWER5 mechanism counts the number of cycles with an empty ROB in the branch misprediction stall counter.
- The second reason for a commit stall is that the instruction at the head of the ROB cannot be committed for some reason. The zero-commit cycle can be attributed to one of the following.
  - The instruction at the head of the ROB is stalled because of a D-cache miss or a D-TLB miss. This causes the D-cache or

D-TLB commit stall counter to be incremented every cycle until the memory operation is resolved.

– The instruction at the head of the ROB is an instruction with latency greater than one cycle, such as a multiply, divide, or a long latency floating-point operation, and the instruction has not yet completed. The long-latency commit stall counter is incremented every cycle until the commit stage can again make progress.

These three CPI stack building approaches, the two naive approaches and the more complex IBM POWER5 approach, are both built in a bottom-up fashion by focusing on individual events that affect performance, such as cache misses, without considering how to combine individual counts into a comprehensive picture of cycle components. Our approach in contrast is constructed top-down, by first developing a performance model, i.e., interval analysis, and then designing a set of performance counters that can accurately measure the penalties as defined by the performance model. As the next section will show, this leads to much more accurate cycle component stacks.

5.3 Validation

The counter architecture described above is by construction able to exactly compute the miss event penalties as defined by interval analysis (considering the initial FMT design). The first part in this validation section compares the individual miss event penalties to the penalties assumed by other methods. The second part will evaluate the overall accuracy compared to simulation. The processor configuration considered is the baseline configuration (see Appendix A).

5.3.1 Instruction cache misses

Figure 5.4 shows the individual penalty per L1 instruction cache miss for our cycle accounting architecture compared to the other discussed methods (we did not include a bar for the naive_non_spec approach, because the individual penalties are equal to the naive approach; they differ only in the number of instruction cache misses considered). This figure shows that the naive method computes the instruction cache miss penalty in an accurate way: the penalty of an instruction cache miss

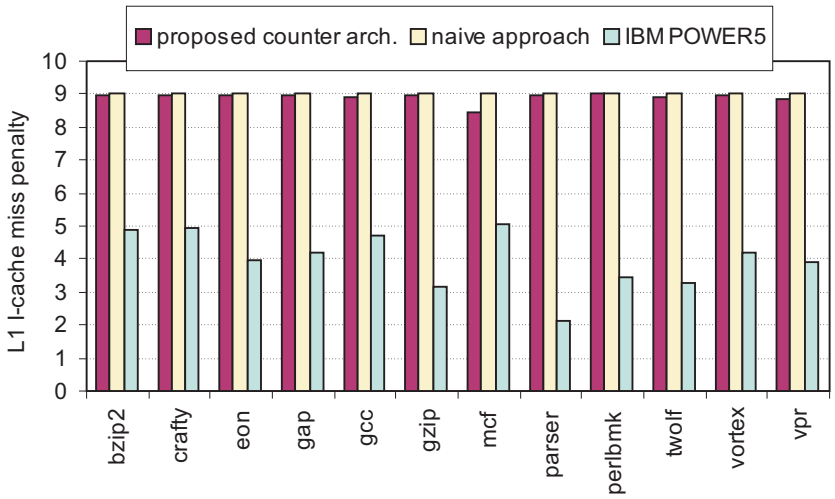

Figure 5.4: The average penalty per L1 I-cache miss.

is equal to the access latency of the appropriate cache level. The IBM POWER5 model in contrast only counts the number of cycles commit is stalled due to the ROB being empty after an instruction cache miss. Looking back at Figure 3.3 (page 28), this assumption is a substantial underestimation of the actual penalty. The time needed to drain the instruction window is not counted in the IBM POWER5 mechanism, whereas it is an important part of the instruction cache miss penalty. Note that if the drain time takes longer than the miss latency (in case a largely filled ROB needs to be drained and there is low ILP or a significant fraction of long-latency instructions), the IBM POWER5 approach will not ascribe any penalty cycles to the instruction cache miss.

5.3.2 Branch mispredictions

The average cycle penalty for a branch misprediction is illustrated in Figure 5.5. Remember that the penalty of a branch misprediction consists of the branch resolution time and the front-end refill time. It is therefore substantially larger than the front-end pipeline length as assumed by the naive approach. The penalty estimation of the IBM POWER5 approach is even worse, because the number of cycles the commit stage is stalled due to an empty ROB after a branch misprediction is usually smaller than the front-end refill time.

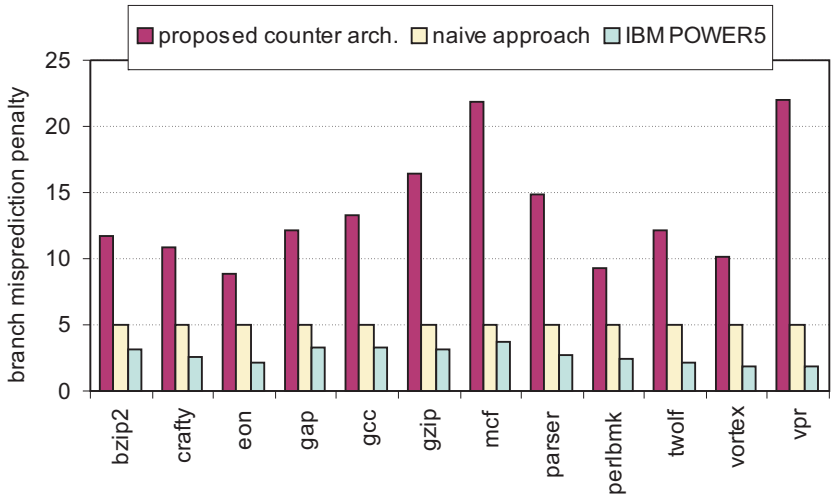

Figure 5.5: The average penalty per branch misprediction.

5.3.3 Short back-end misses

Short back-end misses (L1 data cache misses and medium-latency functional units) rarely show a penalty. This is because dispatch is only stalled when the ROB is filled up completely, which does not occur often on short misses. The naive approach, penalizing each L1 data cache miss as the full access time to the L2 cache, largely overestimates the penalty, see Figure 5.6. The IBM POWER5 mechanism performs better by only counting the number of cycles commit is stalled due to short back-end misses. It starts counting when the miss is at the head of the ROB, which is however earlier than what interval analysis defines as the penalty, namely only the number of cycles the ROB is actually full. Therefore the IBM POWER5 approach still overestimates the impact of short back-end misses.

5.3.4 Long back-end misses

Figure 5.7 shows the average penalty per long-latency load miss. Interval analysis showed that the penalty of an isolated long latency load miss is approximately equal to the main memory latency, but that several long-latency load misses can overlap, reducing the average penalty. The naive approach does not take into account these overlaps

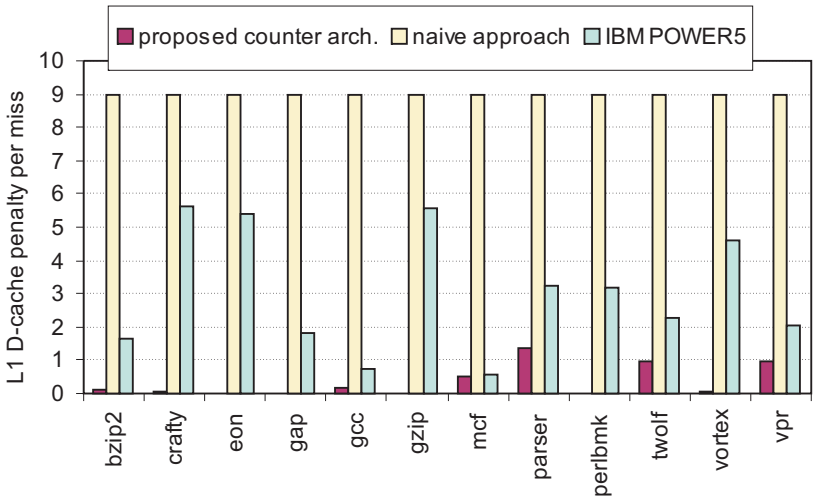

Figure 5.6: The average penalty per L1 D-cache miss.

and therefore overestimates the total penalty. The IBM POWER5 mechanism naturally handles these overlap effects, as does our performance counter architecture, by only looking at the first load miss that hits the head of the ROB. The slight overestimation in the IBM POWER5 approach due to not waiting until the ROB is completely full is negligible compared to the full penalty.

To summarize we can say that instruction cache miss penalties are accurately measured by the naive method, but not by the IBM POWER5 approach; that back-end miss penalties are well estimated by the IBM POWER5 method, but not by the naive approach; and that both approaches fail to characterize the penalty of a mispredicted branch accurately.

5.3.5 Overall accuracy

This section evaluates the overall accuracy of the proposed hardware performance counter architecture. We compare our two hardware implementations, FMT and sFMT, against the IBM POWER5 mechanism, the naive and naive_non_spec approaches and two simulation-based cycle stacks.

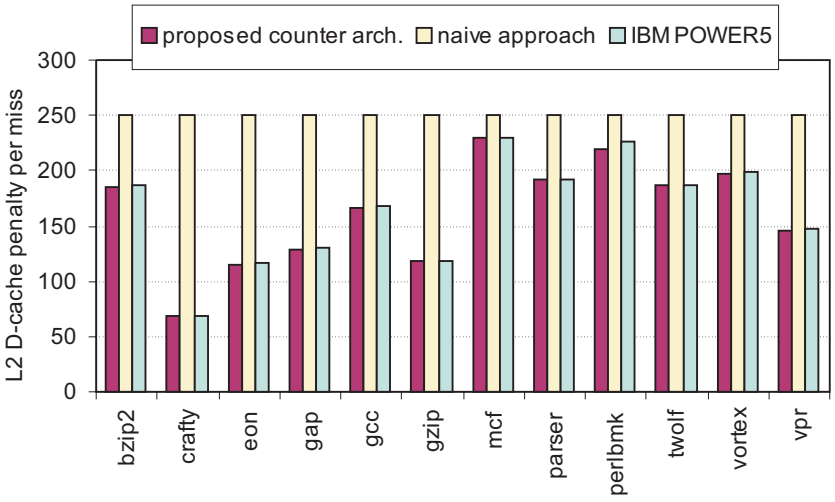

Figure 5.7: The average penalty per L2 D-cache miss.

The simulation-based cycle stacks will serve as a reference for comparison. We use two simulation-based stacks because of the difficulty in defining what a standard 'correct' cycle stack should look like. In particular, there will be cycles that could reasonably be ascribed to more than one miss event. Hence, if we evaluate cycle stack values in one sequence, we may get different numbers than if they are evaluated in a different sequence. To account for this effect, two simulation-based cycle stacks are generated as follows. We first run a simulation assuming perfect branch prediction and perfect caches, i.e., all branches are correctly predicted and all cache accesses are L1 cache hits. This yields the number of cycles for the base cycle component. We subsequently run a simulation with a real L1 data cache. The additional cycles over the first run (which assumes a perfect L1 data cache) gives the cycle component due to L1 data cache misses. The next simulation run assumes a real L1 data cache and a real branch predictor; this computes the branch misprediction cycle component. For computing the remaining cycle components, we consider two sequences. The first sequence is the following: L1 I-cache, L2 I-cache, I-TLB, L2 D-cache and D-TLB; the second sequence, called the 'inverse order', first computes the L2 D-cache and D-TLB components and then computes the L1 I-cache, L2 I-cache and I-TLB CPI components. Our simulation results show that the sequence in which the cycle components are computed only has a

small effect on the overall results. This follows from the small percentages of cycles that process overlapping front-end and back-end miss event penalties, as previously shown in Table 3.1.

Figures 5.8 and 5.9 show normalized cycle stacks for the SPECint 2000 benchmarks for the simulation-based approach, the naive and naive_non_spec approach, the IBM POWER5 approach, and the proposed FMT and sFMT approaches. Figure 5.10 summarizes these cycle stacks by showing the maximum cycle component errors for the various cycle stack building methods.

Figures 5.8 and 5.9 show that the naive approach results in cycle stacks that are highly inaccurate (and not even meaningful) for some of the benchmarks. The sum of the miss event counts times the miss penalties is larger than the total cycle count; this causes the base cycle component, which is the total cycle count minus the miss event cycle count, to be negative. This is the case for a number of benchmarks, such as gap, gcc, mcf, twolf and vpr, with gcc being the most notable example. The reason why the naive approach fails in building accurate cycle stacks is that the naive approach does not adequately deal with overlapped long back-end misses, does not accurately compute the branch misprediction penalty, and in addition, it counts I-cache (and I-TLB) misses along mispredicted paths. However, for benchmarks that have very few overlapped back-end misses and very few I-cache misses along mispredicted paths, the naive approach can be fairly accurate, see for example eon and perlbmk. The naive_non_spec approach, which does not count miss events along mispredicted paths, is more accurate than the naive approach, however, the cycle stacks are still not very accurate compared to the simulation-based cycle stacks.

The IBM POWER5 approach clearly is an improvement compared to the naive approaches. For the benchmarks where the naive approaches resulted in a negative base component, the IBM POWER5 mechanism succeeds in producing meaningful cycle stacks. However, compared to the simulation-based cycle stacks, the IBM POWER5 cycle stacks are still inaccurate, see for example crafty, eon, gap, gzip, perlbmk and vortex. The reason the IBM POWER5 approach falls short is that the IBM POWER5 mechanism underestimates the I-cache miss penalty as well as the branch misprediction penalty.

The FMT and sFMT cycle stacks track the simulation-based cycle stacks very closely. Whereas both the naive and IBM POWER5 mechanisms show high errors for several benchmarks, the FMT and sFMT

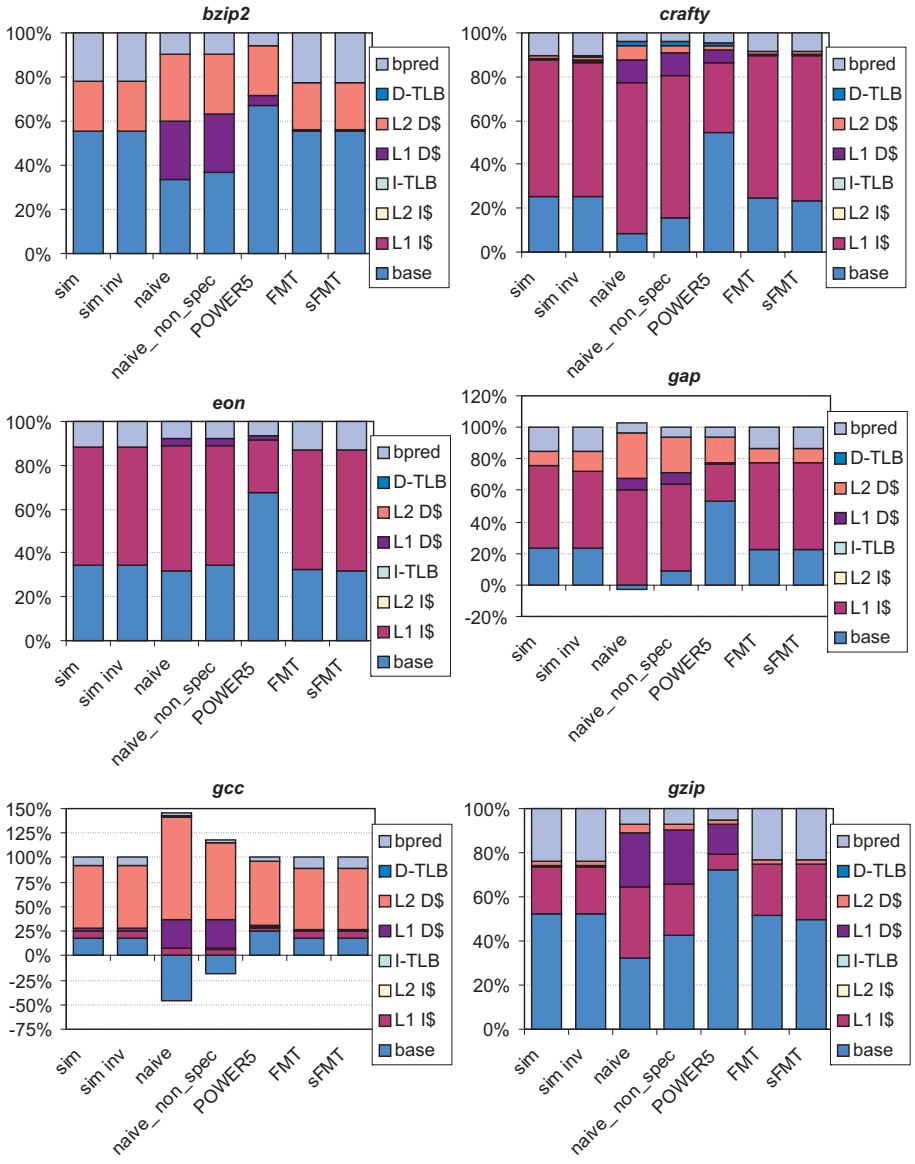

Figure 5.8: Normalized cycle stacks for the SPECint2000 benchmarks: the simulation-based approach, the inverse order simulation-based approach, the naive approach, the naive_non_spec approach, the IBM POWER5 approach and the FMT and sFMT approaches.

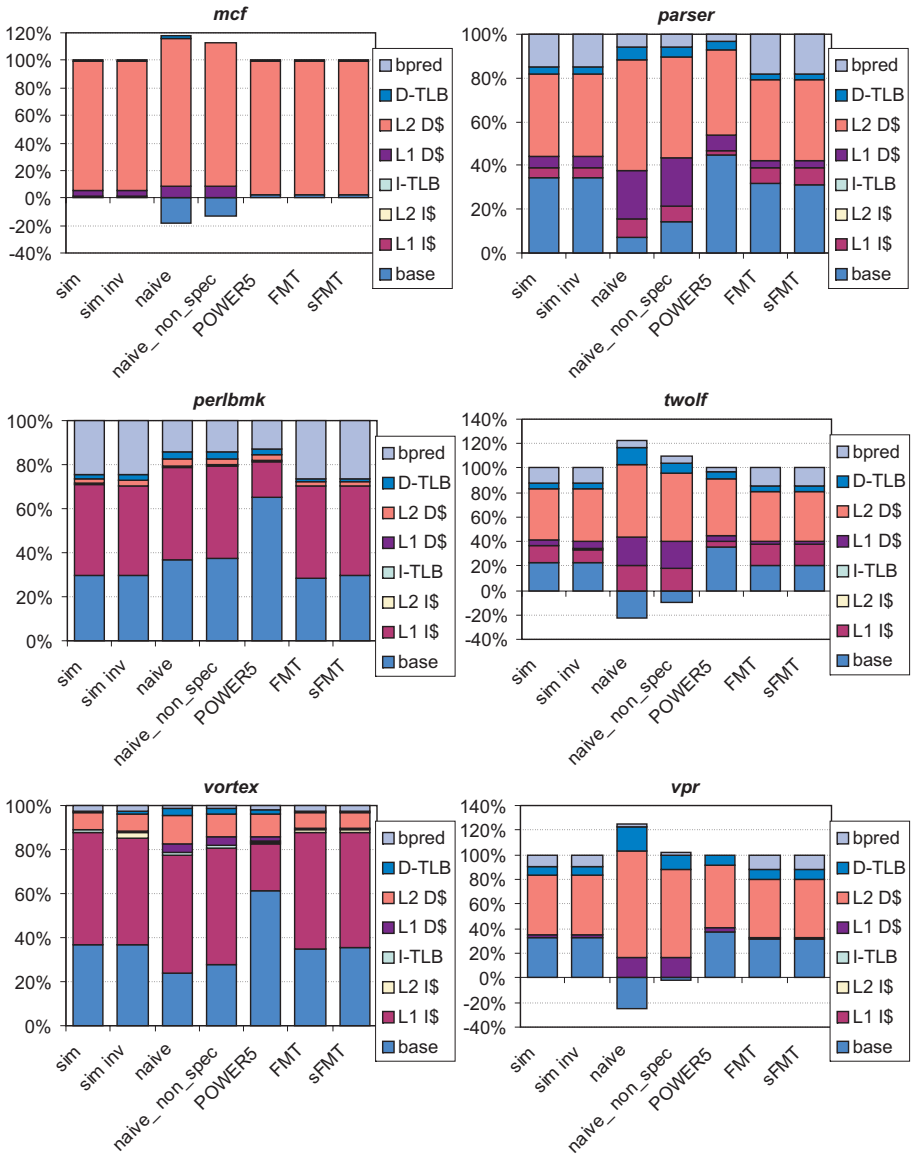

Figure 5.9: Normalized cycle stacks for the SPECint2000 benchmarks (continued)

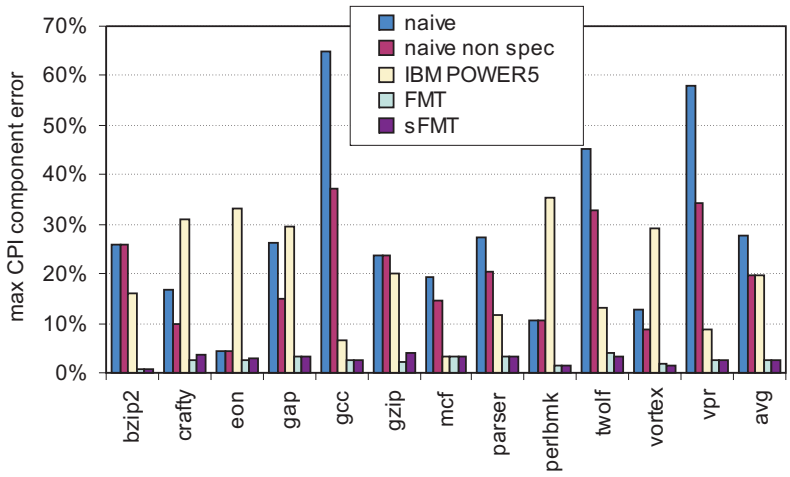

Figure 5.10: Maximum cycle component error for the naive approaches, the IBM POWER5 approach, FMT and sFMT compared to the simulation-based cycle stacks.

architectures show significantly lower errors for all benchmarks. All maximum cycle component errors are less than 4%, see Figure 5.10. The average error for FMT and sFMT is 2.5% and 2.7%, respectively.

5.4 Related work

The Intel Itanium processor family provides a rich set of hardware performance counters for computing cycle component stacks [47]. These hardware performance monitors effectively compute the number of lost cycles under various stall conditions such as branch mispredictions, cache misses, etc. The Digital Continuous Profiling Infrastructure (DCPI) [3] is another example of a hardware performance monitoring tool for an in-order architecture. Computing cycle component stacks for in-order architectures, however, is relatively easy compared to computing cycle component stacks on out-of-order architectures.

Besides the IBM POWER5 mechanism, other hardware profiling mechanisms have been proposed in the recent past for out-of-order architectures. However, the goal for those methods is quite different from ours. Our goal is to build simple and easy-to-understand cycle component stacks, whereas the goal for the other approaches is detailed per-

instruction profiling. For example, the *ProfileMe* framework [18] randomly samples individual instructions and collects cycle-level information on a per-instruction basis. Collecting aggregate cycle component stacks can be done using the ProfileMe framework by profiling many randomly sampled instructions and by aggregating all of their individual latency information. An inherent limitation with this approach is that per-instruction profiling does not allow for modeling overlap effects. The ProfileMe framework partially addresses this issue by profiling two potentially concurrent instructions. Shotgun profiling [35] tries to model overlap effects between multiple instructions by collecting miss event information within hot spots using specialized hardware performance counters. A post mortem analysis then determines, based on a simple processor model, the amount of overlaps and interactions between instructions within these hot spots. Per-instruction profiling has the inherent limitation of relying on (i) sampling which may introduce inaccuracy, (ii) per-instruction information for computing overlap effects, and (iii) interrupts for communicating miss event information from hardware to software which may lead to overhead and/or perturbation issues.

5.5 Application: studying the performance impact of compiler optimizations

To illustrate the value of being able to construct accurate cycle component stacks, we will show their use in studying the performance impact of compiler optimizations [31]. An optimizing compiler implements a large number of individual optimizations which not only interact with the microarchitecture, but also interact with each other. These interactions can be constructive (improved performance), destructive (lost performance), or neutral. Furthermore, whether there is performance gain or loss often depends on the particular program being optimized and executed.

In practice, the only way that the performance gain (or loss) for a given compiler optimization can be determined is by running optimized programs on the hardware and timing them. This method, while useful, does not provide insight regarding the underlying causes for performance gain/loss. Therefore we will use the proposed hardware performance counter architecture for constructing cycle component stacks to study the impact compiler optimizations have on perfor-

Table 5.1: The SPEC CPU2000 C benchmarks, their inputs and the dynamic instruction count when compiled using the `-O3` optimization flag (in millions).

benchmark	input	dyn. I-cnt (M)
bzip2	lgred.program	2,102
crafty	lgred	781
gap	lgred	672
gcc	lgred.cp-decl.i	4,576
gzip	lgred.graphic	1,682
mcf	lgred	659
parser	lgred	3,944
perlbmk	lgred.makerand	1,943
twolf	lgred	1,236
vortex	lgred	1,256
vpr	lgred route	643
ammp	lgred	1,344
art	lgred	2,038
equake	lgred	817
mesa	lgred	1,691

mance. By doing so, we gain insight into the underlying mechanisms by which compiler optimizations affect out-of-order processor performance. We also compare the results to the performance impact of the same compiler optimizations on an in-order processor.

5.5.1 Experimental setup

For this study, we selected the C benchmarks from the SPEC CPU2000 benchmark suite, see Table 5.1. Because we want to run all the benchmarks to completion for all the compiler optimizations, we use the `lgred` inputs provided by MinneSPEC [59]. The dynamic instruction count of the `lgred` input varies between several hundreds of millions of instructions and a number of billions of instructions.

All the benchmarks were compiled using `gcc` v4.1.1 (dated May 2006) on an Alpha 21264 processor machine. We chose the `gcc` compiler because, in contrast to the native Compaq `cc` compiler, it comes with a rich set of compiler flags that can be set individually. This enables us to consider a wide range of optimization levels. The 22 optimization levels considered in this study are given in Table 5.2. This ordering of optimization settings is inspired by `gcc`'s `-O1`, `-O2` and `-O3` optimization levels; the compiler optimizations are applied on top

Table 5.2: Compiler optimization levels considered in this study.

Abbreviation	*Description*
base	base optimization level: `-O1 -fnotree-ccp -fno-tree-dce -fno-tree-dominator-opts -fno-tree-dse -fno-tree-ter -fno-tree-lrs -fno-tree-sra -fno-tree-copyrename-fno-tree-fre -fno-tree-ch -fno-cprop-registers -fno-merge-constants -fno-loop-optimize -fno-if-conversion -fno-if-conversion2 -fno-unit-at-a-time`
basic tree opt	basic optimizations on intermediate SSA code tree
const prop / elim	merge identical constants across compilation units constant propagation and copy elimination
loop opt	loop optimizations: move constant expressions out of loop and simplify exit test conditions
if-conversion	if-conversion: convert control dependencies to data dependencies using predicated execution through conditional move instructions
O1	optimization flag `-O1`
O2 -fnoO2	`-O2` with all individual `-O2` optimization flags disabled
CSE	apply common subexpression elimination
BB reorder	reorder basic blocks in order to reduce the number of taken branches and improve code locality
strength red	strength reduction optimization and elimination of iteration variables
recursion opt	optimize sibling and tail recursive function calls
instruction scheduling	reorder instructions to eliminate stalls due to required data being unavailable; includes scheduling instructions across basic blocks and is specific for target platform on which the compiler runs
strict aliasing	assumes that an object of one type never resides at the same address as an object of a different type, unless the types are almost the same
alignment	align the start of branch targets, loops and functions to a power-of-two boundary
adv tree opt	advanced intermediate code tree optimizations
O2	optimization flag `-O2`
aggr loop opt	perform more aggressive loop optimizations
inlining	integrate simple functions into their callers
O3	optimization flag `-O3`
loop unroll	unroll loops whose number of iterations can be determined at compile time or upon entry to the loop
software pipelining	modulo scheduling
FDO	feedback-directed optimization using edge couts

of each other to progressively evolve from the base optimization level to the most advanced optimization level. The reason for working with these optimization levels is to keep the number of optimization combinations at a tractable number while exploring a wide enough range of optimization levels—the number of possible optimization settings by setting individual flags is obviously huge and impractical to do. We believe the particular ordering of optimization levels does not affect the overall conclusions from this study.

We simulated all the binaries on the SimpleScalar simulator augmented with the cycle accounting performance counter architecture, using the baseline processor configuration in Appendix A. We also simulated them on a similar in-order configuration. Cycle accounting on an in-order processor is much more simple than on an out-of-order processor. Since an in-order processor has no instruction window, we focus on issue instead of dispatch to perform cycle accounting. When no instruction can be issued in a particular cycle, the mechanism increments the count of the appropriate cycle component. For example, when the next instruction to issue stalls for a register to be produced by an L2 miss, the cycle is assigned to the L2 D-cache miss cycle component. Similarly, if no instructions are available in the pipeline to issue because of a branch misprediction, the cycle is assigned to the branch misprediction cycle component. Since misspeculated instructions are never issued (the branch will be resolved first, due to the in-order execution), we do not have to take into account misses along mispredicted paths, and the branch penalty only consists of the front-end refill time.

5.5.2 Out-of-order processor performance

Before discussing the impact of compiler optimizations on out-of-order processor performance in great detail on a number of case studies, we first present and discuss some general findings.

Figure 5.11 shows the average normalized execution time for the sequence of optimizations used in this study. The horizontal axis shows the various optimization levels; the vertical axis shows the normalized execution time (averaged across all benchmarks) compared to the base optimization setting. On average, over the set of benchmarks and the set of optimization settings considered in this study, performance improves by 15.1% compared to the base optimization level. (Note that our base optimization setting already includes a number of optimiza-

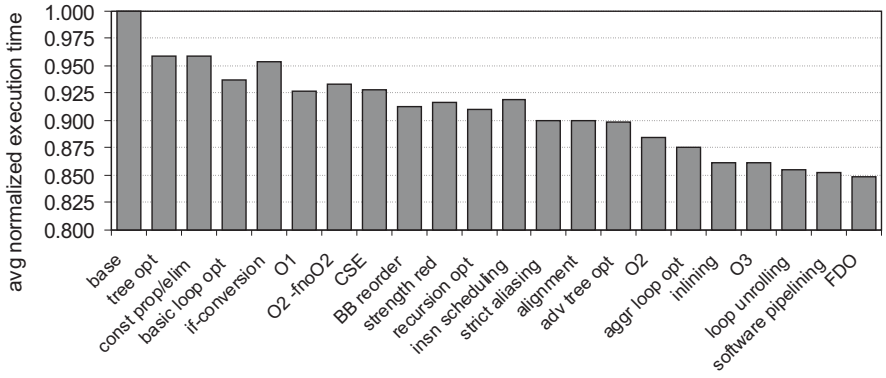

Figure 5.11: Averaged normalized execution time on a superscalar out-of-order processor.

tions, and results in 40% better performance than the `-O0` compiler setting.) Some benchmarks, such as ammp and mesa observe no or almost no performance improvement. Other benchmarks benefit substantially, such as mcf (19%), equake (23%) and art (over 40%).

Figure 5.12 summarizes the total performance improvement for the individual cycle components. This graph divides the total 15.1% performance improvement by the contributions in each of the cycle components.

The MLP and branch penalty components are no explicit cycle stack components, but in order to make the analysis more interesting, we split up the L2 D-cache miss and D-TLB miss component improvements into (i) the improvement due to a decrease in the number of miss events, and (ii) the improvement due to an increase in the exploited memory-level parallelism. We do so by also counting the decrease (or increase) of the number of miss events, and comparing them with the cycle component improvement. The part of the cycle component improvement that is proportional to the decrease of the number of miss events is accounted to the 'no / L2 D-cache' and 'no / D-TLB miss' component, and the remaining part is accounted to the 'MLP' component.

Likewise, we split up the branch misprediction cycle component into a component that indicates the improvement due to the decrease of the number of branch mispredictions, and a component that indicates the improvement due to the decrease in the average branch misprediction penalty (i.e., due to a smaller branch resolution time). Again, this

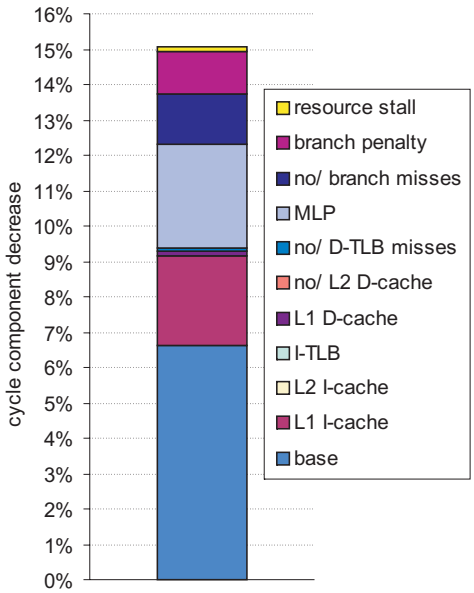

Figure 5.12: Overall performance improvement on an out-of-order processor across the various compiler settings partitioned by cycle component.

is done by dividing the total branch misprediction cycle component improvement into a part that is proportional to the decrease of the number of branch mispredictions, and the remaining part, that is due to smaller branch misprediction penalties.

The 'resource stall' component reflects the performance improvement in the number of cycles the processor is stalled in the absence of miss events, i.e., for the out-of-order processor, when the ROB fills in the absence of miss events (due to very long critical dependency paths), and for the in-order processor, when the next instruction cannot issue because it depends on an instruction that is not yet completed and that is not experiencing a miss event

There are a number of interesting insights to be gained from the above analysis concerning the impact of compiler optimizations on out-of-order processor performance.

First, compiler optimizations reduce the dynamic instruction count and improve the base cycle component. Figure 5.12 shows that an absolute 6.6% performance improvement (or 43.9% of the total improvement) comes from reducing the base cycle component. As such, we

conclude that reducing the dynamic instruction count, which has been a traditional objective for optimization dating back to sequential (non-pipelined) processors, is still an important optimization criterion for today's out-of-order processors.

Compiler optimizations that aim at improving the critical path of inter-operation dependencies only improve the branch misprediction penalty. This is a key new insight from interval analysis: the critical path of inter-operation dependencies is only visible through the branch misprediction penalty and by consequence, optimizations targeted at reducing chains of dependent instructions only affect the branch resolution time; on a balanced processor, inter-operation dependencies not residing on the critical path leading to a mispredicted branch can be effectively hidden by out-of-order execution. Note that optimizations targeting the inter-operation critical path may also improve the 'resource stall' cycle component; in practice though, the performance improvement due to this effect is negligible (less than 0.2% on average). Figure 5.12 shows the improvement in the branch resolution time across the optimization settings; this is a 1.2% absolute improvement or a 7.8% relative improvement.

Finally, compiler optimizations significantly affect the number of miss events and their overlap behavior. According to Figure 5.12, 9.6% of the total performance improvement comes from a reduced number of branch mispredictions, and 16.7% of the total performance improvement comes from improved L1 I-cache behavior. 19.5% of total performance improvement is due to improved memory-level parallelism (MLP). In other words, compiler optimizations that bring L2 cache miss loads closer to each other in the dynamic instruction stream improve performance substantially by increasing the amount of MLP.

5.5.3 Compiler optimization analysis case studies

We now present some case studies illustrating the power of interval analysis for gaining insight into how compiler optimizations affect out-of-order processor performance. Figure 5.13 and 5.14 show normalized cycle distributions for individual benchmarks—we selected the benchmarks that are affected most by the compiler optimizations. These bars are computed as follows. For all compiler optimization settings, we compute the cycle counts for each of the nine cycle components: base, L1 I-cache, L2 I-cache, I-TLB, L1 D-cache, L2 D-cache, D-TLB, branch

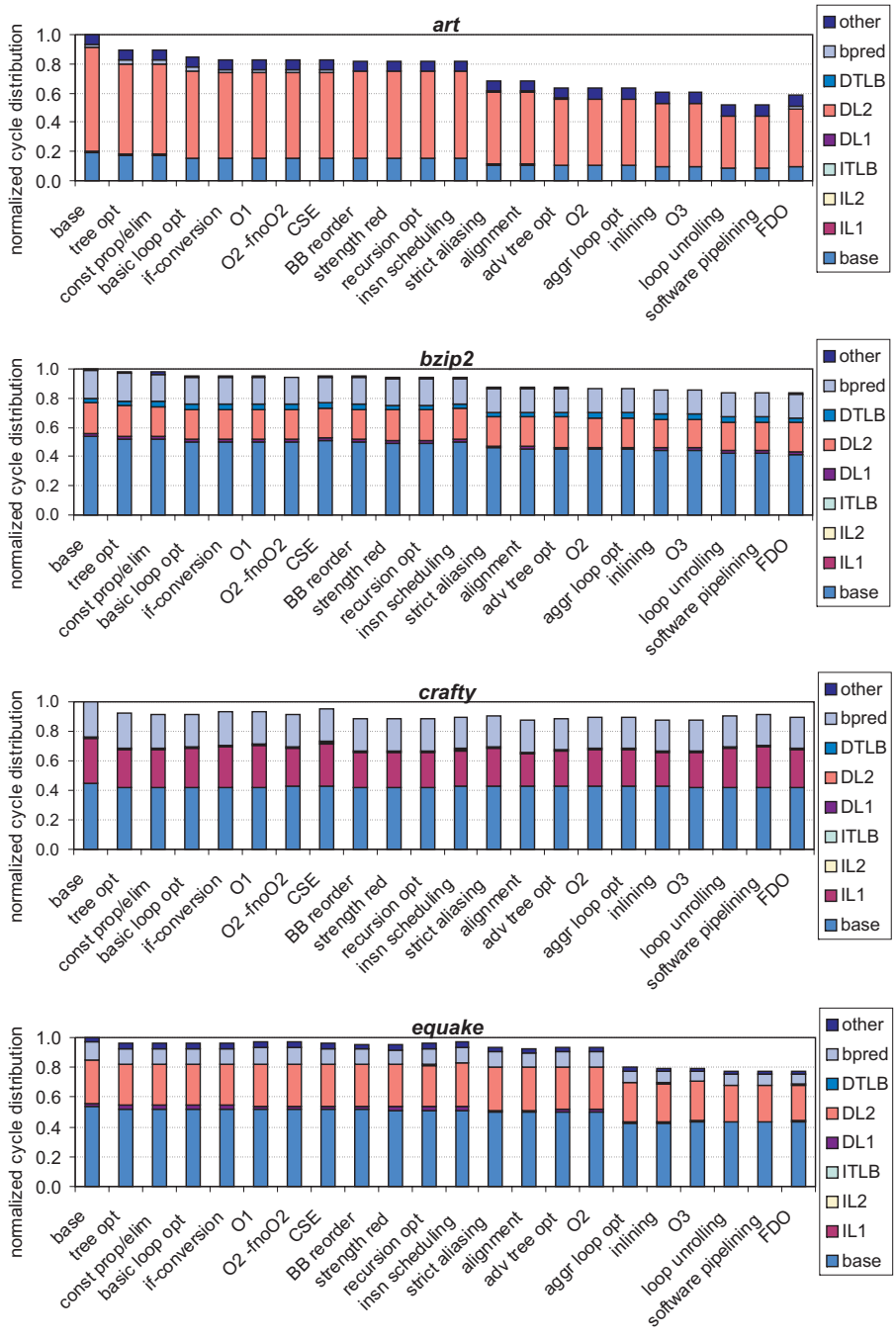

Figure 5.13: Normalized cycle distributions for the out-of-order processor for art, bzip2, crafty and equake.

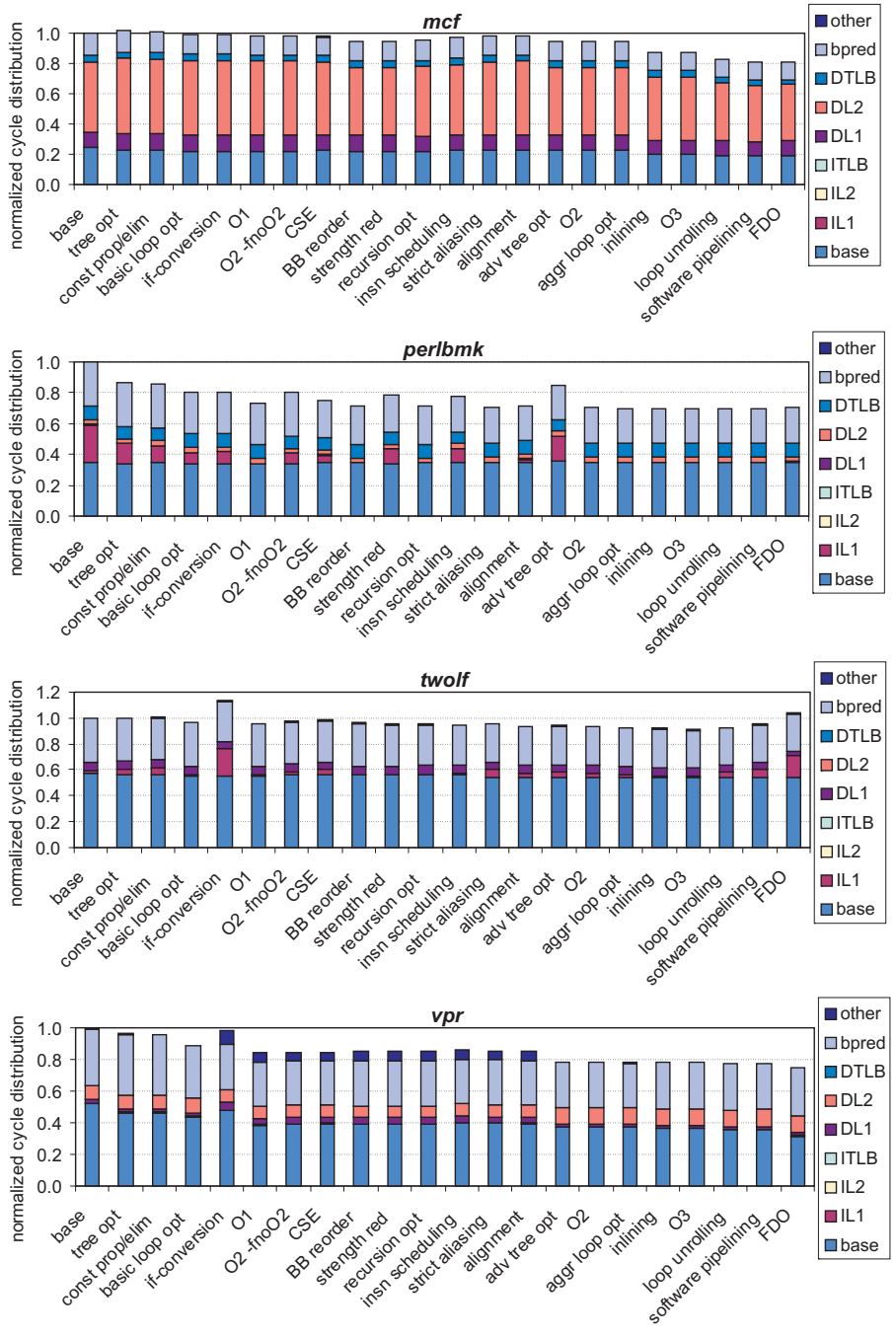

Figure 5.14: Normalized cycle distributions for the out-of-order processor for mcf, perlbmk, twolf and vpr.

Table 5.3: The number of benchmarks (out of 15) for which a given compiler optimization has an positive (more than 0.1%) effect on the various cycle components, the number of retired instructions, the number long back-end misses and their MLP, and the number of branch mispredictions and their penalties. Numbers larger than or equal to 9 are shown in bold.

	cycle components										#insns	DL2 and DTLB misses			br mispreds	
optimization	total	base	IL1	IL2	ITLB	DL1	DL2	DTLB	bpred	other		#DL2	#DTLB	MLP	#bmp	pen
basic tree opt	**11**	**14**	5	0	0	2	6	2	**11**	0	**14**	3	0	5	**11**	3
cst prop /elim	6	2	1	0	0	1	0	0	8	0	7	0	0	1	4	4
loop opt	**12**	**12**	3	0	0	3	3	0	8	8	**13**	3	0	3	3	**9**
if-conversion	7	1	3	0	0	1	6	1	**10**	1	1	2	0	6	8	5
O1	**9**	**10**	4	0	0	2	2	0	3	1	**11**	1	0	2	3	5
O2 -fnoO2	5	0	3	0	0	2	2	1	8	1	0	1	0	1	5	6
CSE	6	5	3	0	0	2	1	1	8	2	**9**	1	1	1	6	6
BB reordering	**10**	**10**	6	0	0	2	2	0	4	1	**11**	2	1	0	6	1
strength red	4	3	1	0	0	0	1	1	3	0	2	1	0	1	2	1
recursion opt	7	4	4	0	0	0	3	0	4	0	6	3	0	0	4	3
insn scheduling	5	1	2	0	0	4	3	1	**10**	4	0	1	0	3	5	**10**
strict aliasing	8	**11**	2	0	0	5	3	1	6	3	**11**	1	1	5	0	**10**
alignment	5	3	2	0	0	0	3	1	5	0	4	2	0	2	4	2
adv tree opt	6	3	3	0	0	2	4	2	4	3	5	1	2	3	3	5
O2	**9**	7	4	0	0	0	2	0	6	0	7	3	0	2	4	4
aggr loop opt	7	2	3	0	0	1	2	0	4	1	3	1	0	1	4	0
inlining	**12**	**10**	3	0	0	0	7	1	**9**	1	**12**	4	1	4	7	4
O3	5	2	2	0	0	0	1	0	3	0	2	0	0	1	3	1
loop unrolling	**9**	**11**	1	0	0	3	4	0	7	2	**12**	0	0	5	3	6
software pipelining	3	1	2	0	0	1	1	0	1	1	0	1	1	0	4	0
FDO	8	7	3	1	0	2	3	1	6	1	**10**	5	2	1	7	5

Table 5.4: The number of benchmarks (out of 15) for which a given compiler optimization has a negative (more than 0.1%) effect on the various cycle components, the number of retired instructions, the number of long back-end misses and their MLP, and the number of branch mispredictions and their penalties. Numbers larger than or equal to 9 are shown in bold.

	cycle components										#insns	DL2 and DTLB misses			br mispreds	
optimization	total	base	IL1	IL2	ITLB	DL1	DL2	DTLB	bpred	other		#DL2	#DTLB	MLP	#bmp	pen
basic tree opt	4	0	1	0	0	2	3	0	3	5	1	2	1	4	1	**10**
cst prop /elim	6	1	4	0	0	0	2	1	2	0	0	2	0	2	3	2
loop opt	1	1	2	0	0	1	3	1	4	0	0	2	1	2	6	3
if-conversion	6	7	3	0	0	2	1	1	1	1	**11**	1	0	2	2	4
O1	4	1	1	0	0	1	3	2	7	2	0	3	1	3	5	4
O2 -fnoO2	8	**11**	3	0	0	0	3	0	3	0	**13**	1	0	2	4	1
CSE	6	6	3	0	0	1	0	1	3	1	4	1	0	1	5	4
BB reordering	2	1	0	0	0	1	3	2	8	2	2	1	1	4	5	**11**
strength red	3	1	1	0	0	0	1	0	1	0	1	0	0	1	0	0
recursion opt	2	1	1	0	0	0	2	1	4	0	0	1	0	2	4	3
insn scheduling	8	**10**	4	0	0	1	5	0	1	1	**11**	3	1	1	3	1
strict aliasing	4	0	2	0	0	0	3	1	2	1	0	3	0	1	6	2
alignment	4	1	4	0	0	2	2	0	3	1	0	2	0	2	3	2
adv tree opt	7	3	3	0	0	0	2	1	7	0	4	2	0	2	6	2
O2	3	1	2	0	0	1	3	1	1	0	2	3	0	1	3	3
aggr loop opt	2	0	1	0	0	1	1	0	1	0	0	1	0	0	1	2
inlining	1	2	2	2	0	3	0	1	4	1	1	1	1	0	3	5
O3	4	1	3	0	0	0	2	0	1	1	1	2	0	0	1	2
loop unrolling	5	1	6	2	0	1	1	0	3	1	1	3	0	1	5	2
software pipelining	3	1	2	0	0	0	1	1	2	0	1	1	0	2	3	4
FDO	6	4	2	0	0	2	3	0	5	2	3	0	0	6	5	7

misprediction and other resource stalls. Once these cycle counts are computed we then normalize the cycle components for all optimization settings to the total cycle count for the base optimization setting.

During the analysis presented in the next discussion we will also refer to Tables 5.3 and 5.4 which show the number of benchmarks for which a given compiler optimization results in a positive or negative effect, respectively, on the various cycle components. These tables also show the number of benchmarks for which the dynamic instruction count is significantly affected by the various compiler optimizations; likewise for the number of long back-end misses and their amount of MLP as well as for the number of branch mispredictions and their penalty. We do not show average performance improvement numbers in these tables because outliers make the interpretation difficult; instead, we treat outliers in the following discussion.

Basic loop optimizations. Basic loop optimizations move constant expressions out of the loop and simplify loop exit conditions. Most benchmarks benefit from these loop optimizations; the reasons for improved performance include a smaller dynamic instruction count which reduces the base cycle component. A second reason is that the simplified loop exit conditions result in a reduced branch misprediction penalty. Two benchmarks that benefit significantly from loop optimizations are perlbmk (6.7% improvement) and art (5.9% improvement). The reason for these improvements is different for the two benchmarks. For perlbmk, the reason is a reduced L1 I-cache component and a reduced branch misprediction component. The reduced L1 I-cache component is due to fewer L1 I-cache misses. The branch misprediction cycle component is reduced mainly because of a reduced branch misprediction penalty—the number of branch mispredictions is not affected very much. In other words, the loop optimizations reduce the critical path leading to the mispredicted branch so that the branch gets resolved earlier. For art on the other hand, the major cycle reduction is observed in the L2 D-cache cycle component. The reason being an increased number of overlapping L2 D-cache misses: the number of L2 D-cache misses remains the same, but the reduced code footprint brings the L2 D-cache misses closer to each other in the dynamic instruction stream which results in more memory-level parallelism.

If-conversion. The goal of if-conversion is to eliminate hard-to-predict branches through predicated execution. The potential drawback of if-conversion is that more instructions need to be executed because instructions along multiple control flow paths need to be executed and part of these will be useless. Executing more instructions reflects itself in a larger base cycle component. In addition, more instructions need to be fetched; we observe that this also increases the number of L1 I-cache misses for several benchmarks (see for example twolf). Approximately half the benchmarks benefit from if-conversion; for these benchmarks, the reduction in the number of branch mispredictions outweights the increased number of instructions that need to be executed. For the other half of the benchmarks, the main reason for the decreased performance is the increased number of dynamically executed instructions.

An interesting benchmark to consider more closely is vpr: its base, resource stall and L1 D-cache cycle components increase by 4.5%, 9.6% and 3.9%, respectively. This analysis shows that if-conversion adds to the already very long critical path in vpr—vpr executes a tight loop with loop-carried dependencies which results in very long dependence chains. If-conversion adds to the critical path because registers may need to be copied using conditional move instructions at the reconvergence point. Because of this very long critical path in vpr, issue is unable to keep up with dispatch which causes the reorder buffer to fill up. In other words, the reorder buffer is unable to hide the instruction latencies and dependencies through out-of-order execution, which results in increased base, L1 D-cache and resource stall cycle components.

Basic block reordering. These compiler optimizations try to minimize the number of taken branches and improve code locality by reordering basic blocks. This is mainly reflected in a decrease in the number of instruction cache misses for some benchmarks (e.g., for crafty and perlbmk). These optimizations also look for branch conditions that are subsumed by previous branch conditions, and eliminates these superfluous branches. This results in a smaller dynamic instruction count for 11 out of 15 benchmarks, see Table 5.3.

Instruction scheduling. Instruction scheduling tends to increase the dynamic instruction count which, in its turn, increases the base cycle component. This observation was also made by Valluri and Govindarajan [99]. The reason for the increased dynamic instruction count is

that spill code is added during the scheduling process by the compiler. Note also that instruction scheduling reduces the branch misprediction penalty for 10 out of 15 benchmarks, see Table 5.3, i.e., the critical path leading to the mispredicted branch is shortened through the improved instruction scheduling. Unfortunately, this does not compensate for the increased dynamic instruction count resulting in a net performance decrease for most of the benchmarks.

Strict aliasing. The assumption that references to different object types never access the same address allows for more aggressive scheduling of memory operations—this is a safe optimization as long as the C program complies with the ISO C99 standard[1]. This results in significant performance improvements for a number of benchmarks, see for example art (16.2%). Strict aliasing reduces the number of non-overlapping L2 D-cache misses by 11.5% for art while keeping the total number of L2 D-cache misses almost unchanged; in other words, memory-level parallelism is increased. Another example is bzip2, where the greatest part of the 7.5% improvement is caused by a reduced number of dynamic instructions (base component). This is because the strict aliasing assumption allows the compiler to find more constant expressions, which can then be moved out of loops.

Inlining. Function inlining is beneficial for most of the benchmarks because of a reduced dynamic instruction count and a reduced number of branch mispredictions, i.e., a smaller number of return address mispredictions. An interesting observation is to be made related to mcf which shows an 8% performance improvement through inlining. The cycle component plot given in Figure 5.14 shows that the reason for the performance improvement is twofold: a reduced dynamic instruction count and a reduced number of (non-overlapping) L2 D-cache and D-TLB misses. The main contributor seems to be the reduced number of non-overlapping L2 D-cache misses. Function inlining eliminates the overhead of the function call which reduces the dynamic instruction count. In addition, the caller can now be optimized more aggressively by including the inlined function in the optimization scope. The end result is that load operations can be scheduled closer to each other which results in improved memory-level parallelism.

[1]The current standard for Programming Language C is ISO/IEC 9899:1999, published 1999-12-01.

Loop unrolling. Loop unrolling improves performance for 9 out of 15 benchmarks. The most important factor is the decrease of the base component. Loop unrolling can reduce the number of executed instructions by combining instructions in the unrolled loop body (e.g., iteration counters can be incremented once per unrolled loop). Another case is the performance improvement of 14% for art. The reason here is the improved memory-level parallelism, i.e., due to a larger degree of freedom to schedule memory operations in the unrolled loop body, more L2 D-cache misses can overlap resulting in a smaller penalty per miss and an improved overall performance.

5.5.4 Comparison with in-order processors

Having discussed the impact of compiler optimizations on out-of-order processor performance, it is interesting to compare against the impact these compiler optimizations have on in-order processor performance. Figure 5.15 shows the average normalized cycle counts on a superscalar in-order processor. Performance improves by 17.5% on average compared to the base optimization level[2]. The most striking observation to be made when comparing the in-order graph (Figure 5.15) against the out-of-order graph (Figure 5.11) is that instruction scheduling improves performance on the in-order processor whereas on the out-of-order processor, it degrades performance. The reason is that on in-order architectures, the improved instruction schedule outweights the additional spill code that may be generated for accommodating the improved instruction schedule. On an out-of-order processor, the additional spill code only adds overhead through an increased base cycle component.

To better understand the impact of compiler optimizations on out-of-order versus in-order processor performance we now compare in-order processor cycle components against out-of-order processor cycle components, see Figure 5.16. To facilitate the discussion, we make the following distinction in cycle components. The first group of cycle components is affected by the dynamic instruction count and the critical path of inter-operation dependencies; these are the base, resource stall, and branch misprediction penalty cycle components. We observe from

[2]Note that this is a relative number, compared to the cycle count for the base optimization level on an in-order processor. In absolute terms, the cycle count on the out-of-order processor is on average approximately 2 times smaller than on the in-order processor.

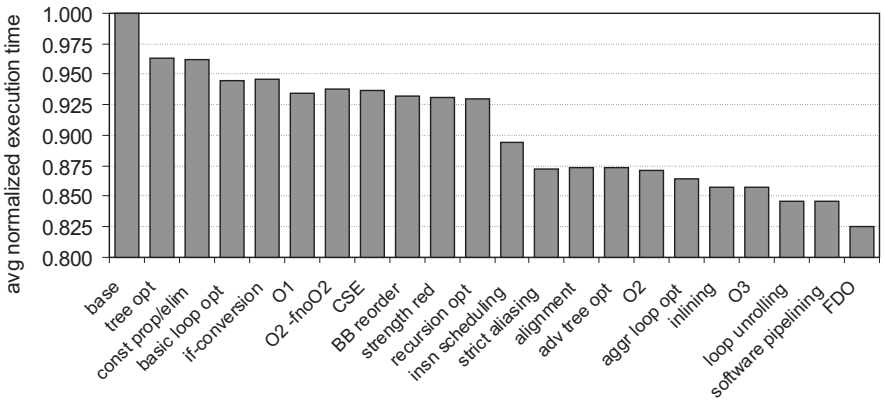

Figure 5.15: Average normalized execution time on a superscalar in-order processor.

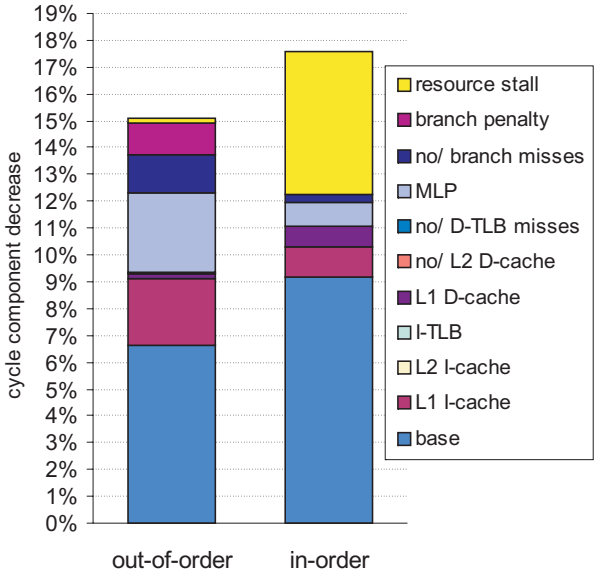

Figure 5.16: Comparing overall performance improvement on an out-of-order processor versus an in-order processor.

Figure 5.16 that these cycle components are affected more by the compiler optimizations for the in-order processor than for the out-of-order processor: 14.6% versus 8.0%. The second group of cycle components are related to the L1 and L2 cache and TLB miss events and the number of branch mispredictions. This second group of cycle components, related to the occurrence of miss events, is affected more for the out-of-order processor: this is only 2.3% for the in-order processor versus 7% for the out-of-order processor. In other words, most of the performance gain through compiler optimizations on an in-order processor comes from reducing the dynamic instruction count and shortening the critical path of inter-operation dependencies. On an out-of-order processor, the dynamic instruction count and the critical path are also important factors affecting overall performance, however, about half of the total performance speedup comes from secondary effects related to I-cache, long-latency D-cache and branch misprediction behavior.

There are three reasons that support these observations. First, out-of-order execution hides part of the inter-operation dependencies and latencies which reduces the impact of critical path optimizations. In particular, in a balanced out-of-order processor, the critical path of inter-operation dependencies is only visible on a branch misprediction. Second, the base and resource stall cycle components are more significant for an in-order processor than for an out-of-order processor; this makes the miss event cycle components *relatively* less significant for an in-order processor than for an out-of-order processor. As such, an improvement to these miss event cycle components results in a smaller impact on overall performance for in-order processors. Third, scheduling instructions can have a bigger impact on memory-level parallelism on an out-of-order processor than on an in-order processor. A good static instruction schedule will place independent long-latency D-cache and D-TLB misses closer to each other in the dynamic instruction stream. An out-of-order processor will be able to exploit the available MLP at run time in case the independent long-latency loads appear within a ROB size from each other in the dynamic instruction stream. An in-order processor on the other hand, may not be able to get to the independent long-latency loads because of the processor stalling on the first instruction that depends on the first long-latency load.

5.6 Summary

In this chapter, we presented a new hardware performance counter architecture that is able to do accurate cycle accounting (i.e., building cycle component stacks) on a single-threaded superscalar out-of-order processor. The performance counter architecture is developed in a top-down fashion, based on interval analysis, and is demonstrated to be far more accurate than previously proposed bottom-up approaches (i.e., starting from existing ad hoc performance counter architectures).

The performance counter architecture uses a front-end miss event table (FMT) to calculate the penalties for the front-end miss events. The FMT measures the instruction cache miss penalties and eliminates instruction cache miss penalties along mispredicted paths. It also keeps track of the branch resolution time of all pending branches, in order to calculate the branch misprediction penalty in case the branch turns out to be mispredicted. Back-end miss event penalties are measured when the ROB is full, as indicated by interval analysis. The resulting cycle component stacks closely track simulation-based cycle component stacks, and are far more accurate than previously proposed mechanisms.

The ability of building accurate cycle component stacks has many possible applications in the development of hardware and software. We elaborated one such application, namely the use of cycle component stacks to study the performance impact of compiler optimizations in detail. This resulted in three main conclusions: (i) reducing the dynamic instruction count remains an important optimization criterion for contemporary out-of-order processors, (ii) critical dependency path optimizations have less impact on an out-of-order processor than on an in-order processor, since they only affect the branch misprediction penalty, and (iii) the reduction of miss events and exploitation of memory-level parallelism is more important for out-of-order processors than for in-order processors.

Chapter 6

Cycle accounting in SMT processors

Computers are like Old Testament gods:
lots of rules and no mercy.
Joseph Campbell

Simultaneous multithreading (SMT) enables a more efficient use of a processor by allowing multiple threads to run concurrently. When one thread experiences a miss event, other threads can still execute instructions, i.e., the performance gaps introduced by miss events (see interval analysis) are filled up with the execution of instructions from other threads, keeping the processor busy. Furthermore, the additional thread-level parallelism provides more independent instructions when the instruction-level parallelism in one thread is lower than the processor width.

There are however some pitfalls. In a simultaneous multithreading processor, many resources are shared and threads interact with each other intensively, which has an impact on the performance of the individual threads. Shared resources can be split up into two different types:

1. Processor core resources, such as fetch bandwidth, instruction window entries, execution units, register file entries, etc. If one thread is allowed to fetch instructions each cycle, the other threads may starve because they did not get their fair share of the fetch bandwidth. Likewise, if one thread occupies most of

the instruction window entries, the other threads can make less progress because less ILP can be extracted.

2. Predictors and caches. Branch predictor entries and cache lines can be shared by different threads, leading to more conflict misses.

The consequence is that the performance of an individual thread depends on the co-executing threads or programs, and performance guarantees cannot be enforced. One thread can push aside all other threads, monopolize all resources and prevent other threads from making any forward progress.

While some fetch policies for SMT processors (see Appendix C) provide a notion of fairness by preventing starving threads and by balancing the performance degradation between the threads, they cannot isolate the performance of the individual threads. Another problem is that current operating systems consider an n-context SMT as an n-core multiprocessor. When different threads are co-scheduled, it is assumed that all threads make equal progress. This is mostly not the case, since the sharing of resources has a substantial impact on the performance of the individual threads. We therefore need a method to determine the actual progress of each of the individual threads during SMT execution.

In this chapter we develop a counter architecture for SMT processors to estimate the progress of individual threads relative to their single-threaded performance, while they are executing on an SMT processor in conjunction with other threads. It is based on the insights provided by interval analysis applied to SMT processor execution. The estimation of the single-threaded progress of the individual threads can then be used by system software or hardware to provide a better quality of service on SMT processors, because it measures the actual progress of each of the co-executing threads. Based on the per-thread progress information, we design a new fetch policy that enforces absolute fairness between threads (i.e., equalizing the performance degradation of each of the threads) or enables performance targets for individual threads (e.g., targetting at a thread reaching 75% of its single-threaded performance, irrespectively of the other co-executing threads).

6.1 Cycle accounting in an SMT processor

Cycle accounting in SMT processors is more complex than in single-threaded processors. The total execution time of an individual thread

consists of (i) cycles where instructions of the thread are executed, (ii) cycles needed to handle the miss events caused by the thread, and (iii) cycles lost due to the co-execution with other threads. To distinguish between those effects, we divide total execution time into three main per-thread cycle components:

- **Base cycle component.** The processor consumes cycles doing computation work for the given thread.
- **Miss event cycle component.** The processor consumes cycles handling miss events caused by the given thread such as cache misses, TLB misses, and branch mispredictions.
- **Waiting cycle component.** The processor consumes cycles for another thread, and therefore cannot make progress for the given thread.

The two first cycle components are equivalent to the base and miss cycle components of the single-threaded cycle accounting method as described in the previous chapter, whereas the third component is specifically introduced by the multi-threading.

We define the per-thread *virtual execution time* as the sum of base and miss event cycle components, i.e., this is the thread's (estimated) total execution time in case the thread would be executed in single-threaded mode. The cycle stack minus the waiting cycle component thus represents an estimate of the single-thread cycle stack. The waiting cycle component quantifies by how much the given thread gets slowed down because of resource sharing in the SMT processor. We initially assume caches and branch predictors to be statically partitioned between the threads such that the threads do not interfere with each other in these structures. By doing so, we consider the proposed counter architecture in isolation, without interfering effects due to shared caches and predictors. However, we will also validate our proposed counter architecture considering fully shared branch predictors and caches.

We now discuss the general mechanisms of per-thread cycle accounting on SMT processors; Section 6.2 then describes its implementation as a counter architecture. We thereby build on the primary observation from the interval model as described in Chapter 3, namely superscalar processor performance can be analyzed in terms of its *dispatch* behavior, which defines the penalties due to miss events. To be able to handle SMT processors that dispatch instructions from multiple threads per cycle, we count *dispatch slots* instead of cycles (i.e., in a

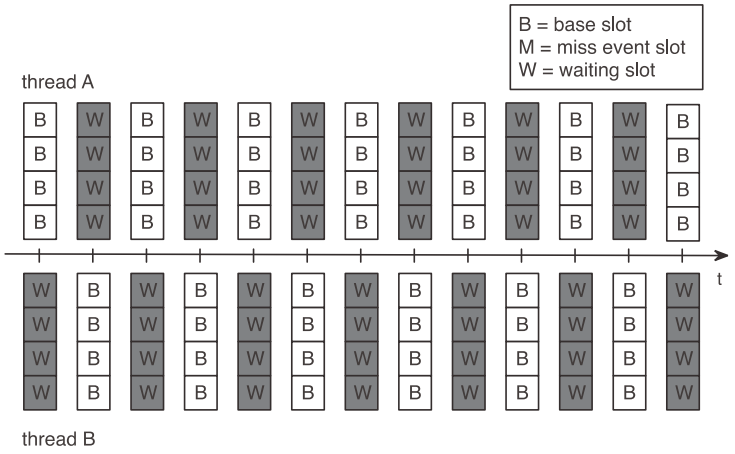

Figure 6.1: SMT processor execution in the absence of miss events.

4-wide dispatch processor, there are 4 dispatch slots per cycle). Cycle components are computed from dispatch slots by dividing the number of dispatch slots by the dispatch width.

6.1.1 No miss events

We first describe SMT performance without any miss events. Figure 6.1 shows the progress for two threads co-executing on an SMT processor in the absence of any miss events. We assume a two-thread processor and a round-robin fetch policy for the purpose of illustration; this does not affect the generality of the formulation though, i.e., the description can be trivially extended to more than two threads and other fetch policies. Each square in the figure represents a dispatch slot. In cycle x, 4 instructions are dispatched from thread A; these slots are accounted as *base* dispatch slots. Thread B on the other hand, cannot make progress; thread B therefore gets 4 *waiting* dispatch slots accounted. In cycle $x + 1$, thread B gets 4 base slots and thread A gets 4 waiting slots, etc. Per-thread performance compared to single-thread performance halves because of SMT execution.

6.1.2 Instruction cache misses

In the event of an L1/L2 I-cache miss or I-TLB miss, as is the case for thread A in Figure 6.2, the processor will no longer be able to dispatch

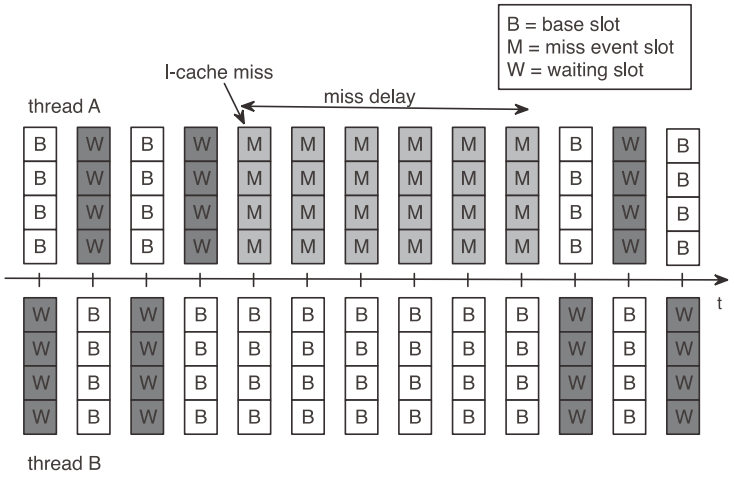

Figure 6.2: SMT processor execution in the presence of an I-cache / I-TLB miss.

instructions for thread A during a number of cycles equal to the miss delay. As in the single-threaded case, the miss delay equals the access time to the next level in the memory hierarchy, and the penalty slots are to be accounted as *miss event* slots.

The other thread, thread B in the example, will be able to dispatch instructions into the pipeline at a rate of 4 instructions per clock cycle. As a result, thread B benefits from the I-cache miss in thread A, i.e., there are no more waiting dispatch slots for thread B under the I-cache miss of thread A—this is where the benefit from SMT execution comes from: thread B can dispatch instructions while the I-cache miss is being resolved for thread A.

6.1.3 Branch mispredictions

Figure 6.3 illustrates what happens upon a branch misprediction for thread A. When thread A dispatches a mispredicted branch, instructions along the incorrect control flow path will enter the processor backend until the branch gets resolved and new instructions along the correct control flow path enter the pipeline. The penalty for the mispredicted branch equals the time between the mispredicted branch being dispatched and correct path instructions being dispatched after the branch resolution, as is the case under single-threaded execution: these dispatch slots are to be counted as *miss event* slots. These miss event

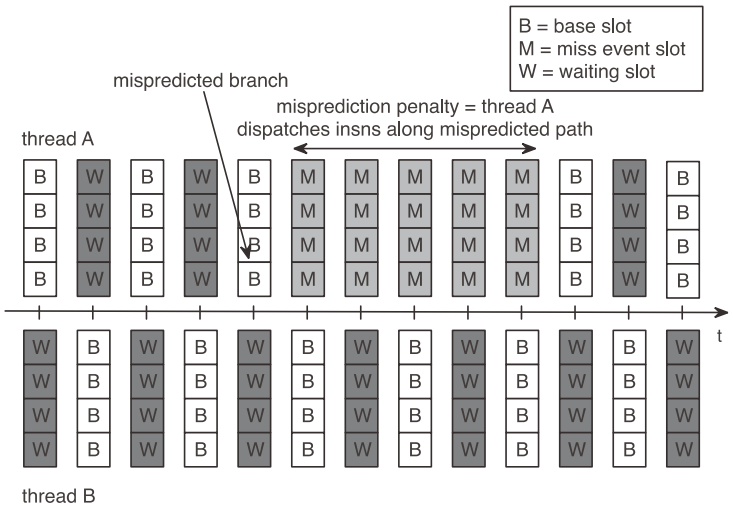

Figure 6.3: SMT processor execution in the presence of a branch misprediction.

slots include the branch resolution time plus the front-end pipeline depth.

The branch resolution time can be affected by the other threads, it can be smaller or longer under SMT execution than under single-threaded execution. The branch resolution time can be slightly smaller due to the fact that in an SMT context in general there are less in-flight instructions per thread compared to single-threaded execution, which shortens the critical path length. It can also be somewhat larger due to execution slots taken by the other threads, slowing down the execution of the dependency path. However, the branch resolution time is mainly dependent on the critical dependency path leading to the branch, meaning that this impact is limited. Using the same branch misprediction penalty definition for multi-threaded performance analysis as in the single-threaded case is a good approximation, which is supported by the results in the evaluation section.

Underneath the penalty for a mispredicted branch, the other thread(s) will continue dispatching instructions; i.e., the performance of thread B is unaffected by the mispredicted branch for thread A, see Figure 6.3.

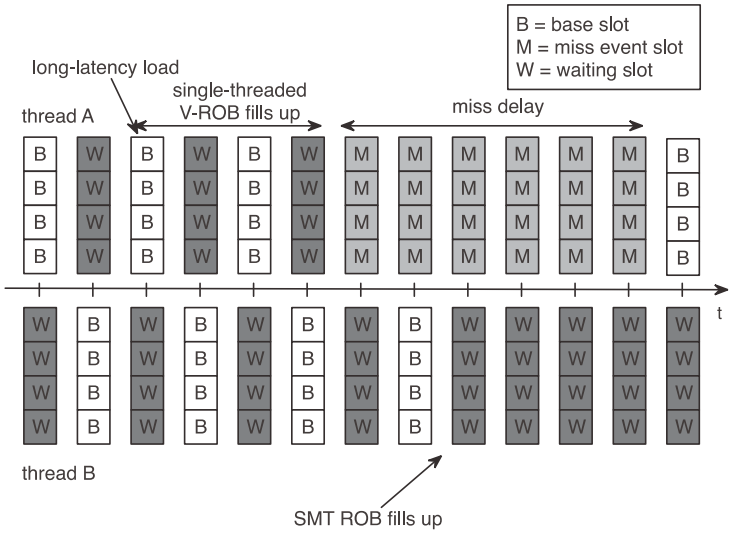

Figure 6.4: SMT processor execution in the presence of a long-latency load miss.

6.1.4 Long-latency load misses

The situation gets more complicated when estimating the penalty due to long-latency loads. Recall that the penalty for a long-latency load under single-threaded execution is the time between the ROB filling up and the miss returning from memory. The key problem now is to estimate when the ROB would fill up under single-threaded execution, during the SMT execution.

We solve this by keeping track of all the base and waiting dispatch slots since the oldest instruction in the reorder buffer for the given thread. The point in time where the number of 'in-flight' base and waiting slots—called the single-thread *virtual ROB (V-ROB)* size—equals the SMT processor's ROB size, signals the point where the ROB would get exhausted under single-threaded execution if this many instructions were dispatched. This assumes that the waiting slots could be used to dispatch instructions from that thread in single-threaded execution. This is illustrated in the example given in Figure 6.4 in which the 16-entry ROB gets virtually exhausted by thread A under single-threaded execution after four cycles. Once this point of V-ROB exhaustion is reached, we start counting miss event slots, i.e., long-latency load miss event dispatch slots.

Since it is possible that the (real) ROB is not yet full when the V-ROB is exhausted, the thread experiencing the long-latency load miss can still dispatch instructions when its V-ROB is full. In that case, we do increment the long-latency load miss component counter, since in single-threaded mode the ROB will be full at that time, but we also increment the number of base dispatch slots and decrement the number of waiting dispatch slots. This is done to ensure that the base component is equal to the number of dispatched instructions. This operation has no impact on the V-ROB size, since waiting slots are exchanged for dispatch slots, and the V-ROB includes both.

Depending on the fetch policy, a long-latency load miss in one thread can eventually cause the ROB to fill up completely and in that case, the other thread(s) (whose V-ROB is not full) are accounted waiting slots, see thread B in the example given in Figure 6.4.

In the above description, we make one implicit assumption: we assume that memory-level parallelism is unaffected by single-threaded versus SMT execution, i.e., the amount of MLP under single-threaded execution is the same as under SMT execution. This is an inaccurate assumption because a single-threaded execution would allocate all the ROB resources and therefore expose more MLP than an SMT execution, where the ROB is shared between the threads. To counter this inaccurate assumption, we introduce an MLP correction mechanism (see Section 6.2.2) that estimates the amount of MLP under single-threaded execution. The ratio of the SMT MLP and single-threaded MLP then serves as a correction factor for the long-latency load miss event slots.

6.1.5 Other resource stalls

Other resource stalls due to long-latency functional units (divide, multiply, etc.), short data cache misses, store buffer stalls, etc., are accounted as miss event slots following the approach for the long-latency loads, i.e., a miss event slot is accounted in case a thread's V-ROB size equals the processor's ROB size.

6.1.6 Stall and flush fetch policies

Some SMT fetch policies stall and / or flush threads to prevent threads from clogging resources (e.g., the flush policy of Tullsen and Brown [96] and the MLP-aware flush policy by Eyerman and Eeckhout [28], see

also Appendix C). In case a thread is stalled by the fetch policy, subsequent dispatch slots are counted as waiting dispatch slots, since in single-threaded mode these slots could have been used to dispatch instructions. When the V-ROB size (that keeps on growing because of the increasing number of waiting slots) equals the machine's ROB size, miss event slots are accounted—this is the point in time where single-threaded execution would fill up the ROB.

In case a thread is (partially) flushed, the number of flushed instructions is subtracted from the base slots counter and added to the waiting slots counter. Flushing instructions has the same result as not having dispatched these instructions, which means that the cycles during which the flushed instructions were dispatched are now similar to waiting cycles. Note that the V-ROB size is unaffected by this operation.

6.2 Implementation

We now provide a functional description of how to implement the cycle accounting mechanism just described into a hardware performance counter architecture. The counter architecture consists of a set of *global* counters for counting per-thread base, miss event, and waiting dispatch slots. We have miss event cycle counters for L1 I-cache, L2 I-cache, L3 I-cache, I-TLB, branch misprediction, L1 D-cache, L2 D-cache, L3 D-cache, D-TLB misses, and other resource stalls such as end-of-cacheline (ECL)[1], write buffer stall (WB), empty dispatch slots because of no-ops, and 'others' (which accounts for the remaining resource stalls, such as the ROB filling up due to long dependency paths). In addition, the counter architecture requires additional hardware structures which we describe next; we again make a distinction between front-end miss events and back-end miss events.

6.2.1 Front-end miss events

To count front-end miss events, we reuse the FMT design as described in the previous chapter. The single adaptation we have to make is to

[1]Since the SMTSIM simulator, which we used in this study, has no fetch buffer, it frequently occurs that fewer instructions than the available width can be dispatched due to taken branches and instructions on different cache lines. If for example only 3 out of 4 instructions can be dispatched due to this effect, we account one dispatch slot to the 'ECL' component.

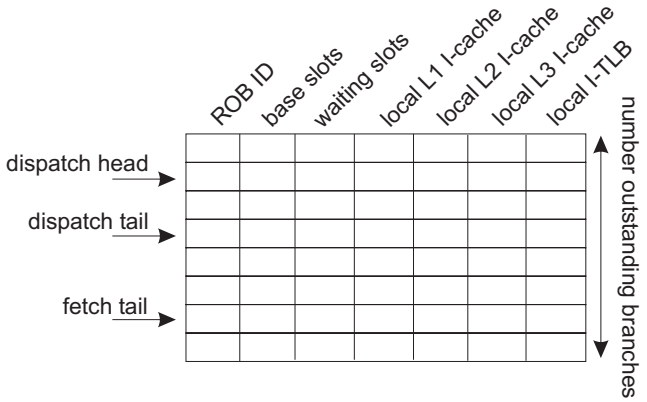

Figure 6.5: Front-end miss event table FMT for SMT cycle accounting.

add counters to differentiate between base and waiting dispatch slots. Each row in the FMT corresponds to an outstanding branch, and counts the number of base, waiting, and miss event (L1 I-cache, L2 I-cache, L3 I-cache and I-TLB) slots prior to this branch, see Figure 6.5. We need one FMT per supported thread context. To save space in the FMT, we decided to eliminate the branch penalty counter, since the branch resolution time can also be found by adding all base, waiting and miss event slots of all rows past the mispredicted branch. When dispatching an instruction, we increment the base cycle counter in the FMT-entry pointed to by the dispatch tail pointer. When an instruction is dispatched for another thread, we increment the waiting cycle counter in the absence of a miss event, and increment the miss event cycle counter in case of a miss event. When committing a branch, the counters of the associated row are added to the respective global counters, and the row is de-allocated.

When resolving a mispredicted branch, the base, waiting and miss event counts of all the subsequent rows are added to the global branch misprediction miss event counter—this is the number of dispatch slots since the mispredicted branch was dispatched, i.e., the branch resolution time—and all dispatch slots hereafter are counted as miss event slots until correct-path instructions are being dispatched; in addition, the subsequent rows are de-allocated.

The hardware cost for the FMT is limited, and requires $c \cdot n \cdot b$ bits in total, with c the number of counters per row, n the number of bits per counter, and b the number of outstanding branches in the processor.

This is $6 \cdot 10 \cdot 16 = 960$ bits in our implementation.

6.2.2 Back-end miss events

For computing the back-end miss event cycles, we need three additional hardware structures.

First, we keep track of the per-thread V-ROB occupancy using a per-thread *In-Flight Slots Counter (IFSC)*, which counts the 'in-flight' base and waiting slots. In addition, we maintain an *In-Flight Waiting Slots (IFWS)* per ROB entry. When dispatching an instruction we initialize the ROB entry's IFWS to zero, and increment the IFSC; the other threads, in the absence of a miss event, increment their IFSCs, as well as the IFWSs pointed to by their ROB tails. When committing an instruction, we decrement the IFSC for the given thread (i.e., there is one dispatch slot less in-flight), and subtract the associated IFWS from the IFSC (i.e., we subtract the waiting slots). We refer to Figure 6.6 for an illustrative example. By doing so, the IFSC counts the per-thread V-ROB occupancy. When the V-ROB occupancy equals the processor's ROB size—the point in time where the single-threaded execution would fill up the ROB—we start counting long-latency miss event slots and other resource stalls as described above.

Second, we need to compute the per-thread MLP under SMT execution. The average number of outstanding long-latency load misses is measured using two counters. The first counter counts the number of cycles for which at least one long-latency load is outstanding; the second counter counts the number of outstanding long-latency loads. The second counter divided by the first yields the average MLP under SMT execution.

Third, we need to estimate the amount of MLP under single-threaded execution. To this end, we introduce a back-end miss event table (BMT) per thread. The BMT has as many rows as the processor supports outstanding long-latency loads, and operates as follows. A BMT row is allocated when committing a long-latency load, and for each committed instruction we update all allocated rows, as described below. Each BMT row holds:

(i) A *Committed Instructions Counter (CIC)* counting the number of instructions committed after the given load miss: this counter is initialized upon the row's allocation and is incremented for each committed instruction.

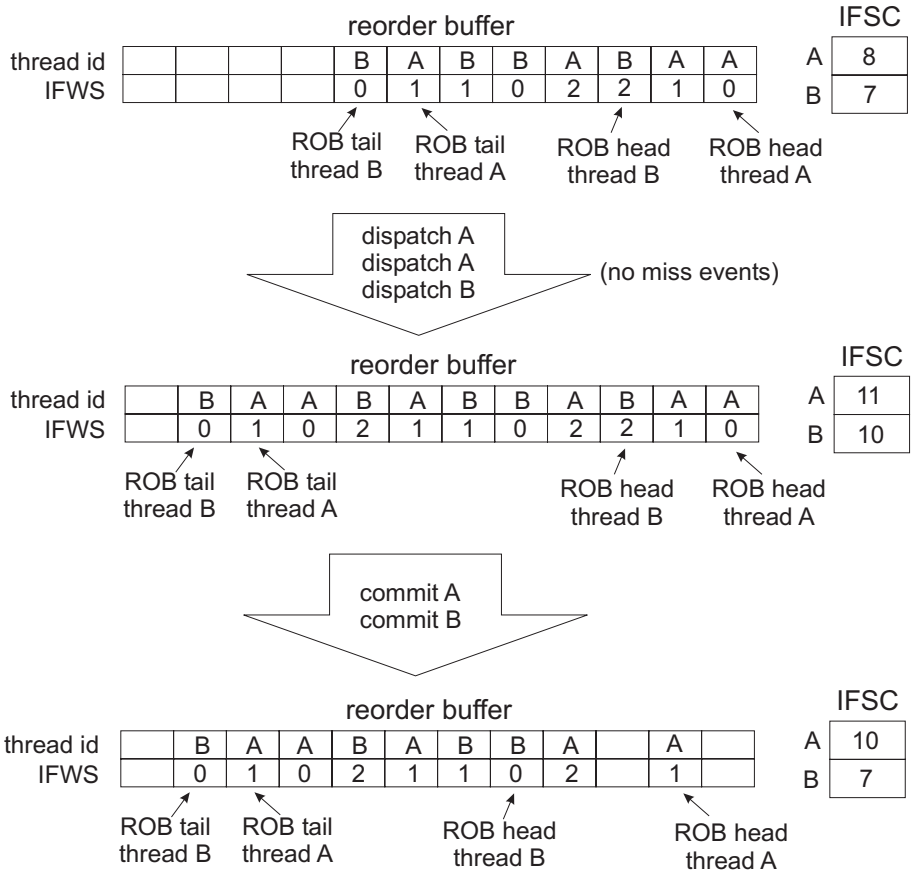

Figure 6.6: Example of IFSC and IFWS operation.
The *top part* indicates the initial situation: thread A and B each have 4 in-flight instructions; thread A has 4 in-flight waiting slots (sum of the IFWSs belonging to thread A) and thread B has 3 in-flight waiting slots, which yields a V-ROB occupancy of 8 for thread A and 7 for thread B, indicated in the corresponding IFSCs. Dispatching an instruction of thread A causes the allocation of a new ROB entry with IFWS set to zero, and an increment of the IFWS pointed to by the ROB tail of thread B (in the absence of miss events).
The *middle part* shows the situation after dispatching two instructions for thread A and one for thread B. The V-ROB occupancies are increased by 3 slots (2 base and 1 waiting for A, and 1 base and 2 waiting for B).
The *bottom part* depicts the situation after committing one instruction of each thread. The IFSC of thread A is decreased by 1 (1 instruction and 0 IFWS), and the IFSC of thread B is decreased by 3 (1 instruction and 2 IFWS).

(ii) An *Output Register Bitmap (ORB)* representing all the architectural registers written by the load and its dependent instructions. The ORB is initialized with the output register of the long-latency load, i.e., a '1' is written at the corresponding bit position, and all the other bits are zeroed. In case a committed instruction depends on a prior long-latency load—this is done by comparing the load's ORB with the instruction's input registers—the load's ORB is updated with a '1' at the bit position corresponding to the committed instruction's output register. In case a committed instruction does not depend on the prior long-latency load, a '0' is written at the bit position corresponding to the committed instruction's output register.

(iii) A *Committed Dependent Instructions Counter (CDIC)* that counts the number of instructions committed that depend on the long-latency load through RAW dependencies—this is computed through the ORBs.

(iv) A *Dependent Load Pointer (DLP)* to the long-latency load that the load depends upon, if applicable—this is also computed using the ORB.

Dispatch would stall in single-threaded execution when the CIC for the long-latency load at the BMT head equals the ROB size, or its CDIC equals the issue buffer size: either the ROB fills up or the issue buffer fills up, respectively. We then count the amount of MLP under single-threaded execution by counting the number of independent long-latency loads in the BMT—this is done by walking the table from head to tail, and counting the long-latency loads that do not depend (using the DLP) on the long-latency load at the BMT head. We subsequently deallocate the entry that initiated the MLP calculation (with a CIC equal to the ROB size or a CDIC equal to the issue buffer size) and all entries that overlap with this long-latency load (i.e., the independent load misses). We have to deallocate these entries, because these load misses will not make it to the head of the ROB in single-threaded execution, so no MLP has to be calculated for them. An example of the operation of the BMT is depicted in Figure 6.7.

The ratio of the MLP under SMT execution with the MLP under single-threaded execution then yields a correction factor for the long-latency load miss cycle component, i.e., dividing the long-latency load

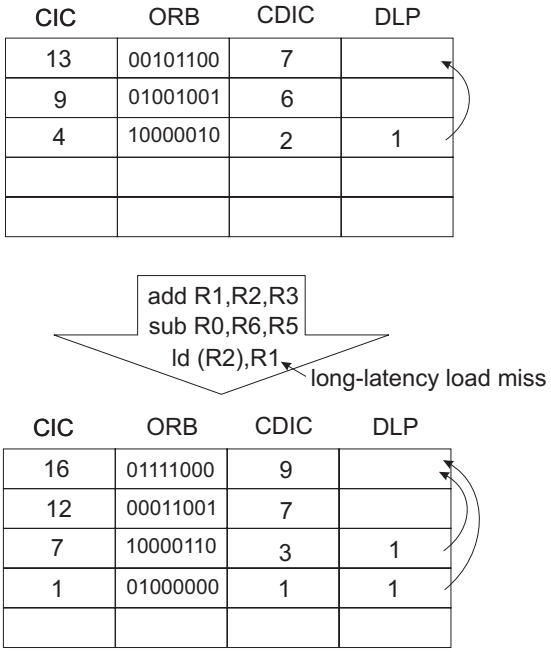

Figure 6.7: Example of BMT operation.
Initially, there are three outstanding load misses. The CICs indicate the number of instructions committed after the load miss (the load miss itself included), the ORBs indicate the registers this load miss produces (from R0 to R7), the CDICs show the number of dependent instructions committed after this load miss, and the DLPs show the dependencies between the load misses (load 3 depends on load 1).
After committing the three instructions shown (the rightmost operands are the destination operands), the BMT of this thread has changed as depicted in the bottom part. 3 instructions have been committed, so the CICs are increased by 3. The first and third instruction depend on the first load, so its CDIC is increased by 2 (and similar for the other CDICs). The ORBs are adapted as described in the text. The third instruction is a long-latency load miss, so a new entry is allocated, and the ORB is initialized with the output register (R1). If the ROB size equals 16 or the issue buffer size equals 9, we should now calculate the MLP for the first load. The MLP equals 2 (first load itself and second load), because the third and fourth load depend on the first. Then the first two entries can be deallocated.

vortex-parser	swim-perlbmk
crafty-twolf	galgel-twolf
facerec-crafty	fma3d-twolf
vpr-sixtrack	apsi-art
vortex-gcc	gzip-wupwise
gcc-gap	apsi-twolf
apsi-mesa	mgrid-vortex
mcf-swim	swim-twolf
mcf-galgel	swim-eon
wupwise-ammp	swim-facerec
swim-galgel	parser-wupwise
lucas-fma3d	vpr-mcf
mesa-galgel	equake-perlbmk
galgel-fma3d	applu-vortex
applu-swim	art-mgrid
mcf-equake	equake-art
applu-galgel	parser-ammp
swim-mesa	facerec-mcf

Table 6.1: The two-thread workloads used in the evaluation.

miss cycle component with this ratio provides an estimate of the long-latency load miss cycle component under single-threaded execution.

The hardware cost for the back-end miss event counter architecture is limited: (i) an IFWS element of a couple bits is needed per ROB entry; (ii) the IFSCs require $\lceil \log_2 ROB \rceil$ bits per thread; (iii) the SMT MLP estimator requires two counters per thread; (iv) the BMT requires $n(2\log_2 ROB + \log_2 n + 64)$ bits in total, with n the maximum number of outstanding long-latency loads, $2\ \log_2 ROB$ bits for the CIC and CDIC counters, $\log_2 n$ bits for the DLP, and 64 bits for the ORB (assuming 64 architectural registers); assuming 16 outstanding long-latency loads and a 256-entry ROB, the BMT incurs 1344 bits of storage.

6.3 Experimental setup

The processor model being simulated is a 4-wide superscalar out-of-order SMT processor, the configuration is described in Appendix A (in the SMT processor section). We use 36 randomly selected two-thread workloads, see Table 6.1, however, most two-thread workloads are memory-intensive in order to stress our cycle accounting architecture, i.e., we found the largest errors to appear for memory-intensive workloads. We fast-forward each benchmark until the beginning of a

200 million instruction early SimPoint is reached, see Appendix A. The default fetch policy is ICOUNT 2.4 [97] which allows up to four instructions from up to two threads to be fetched per cycle.

Our experimental setup was chosen in such a way to stress and evaluate our proposed counter architecture in isolation without interfering effects due to cache and branch predictor sharing. Therefore we assume fair sharing and performance isolation in the memory hierarchy and branch predictor as proposed in prior work [34, 49, 50, 58, 69, 71, 72]. To mimic a fair sharing substrate, we assume a partitioned memory hierarchy and branch predictor in all of our measurements (unless mentioned otherwise), and evaluate the inaccuracy this introduces compared to a shared memory hierarchy and branch predictor (see Section 6.4.4).

6.4 SMT cycle component stacks: evaluation

We now evaluate the accuracy of the per-thread cycle component stacks computed using the proposed counter architecture under SMT execution, called the SMT cycle component stacks, by comparing them against the cycle component stacks computed under single-threaded execution, called the ST cycle component stacks. This is done as follows. We run each two-thread workload for 400 million instructions, compute the per-thread cycle component stacks as described in the previous sections, and record the number of instructions executed so far for each thread. We then run each thread in single-threaded mode for as many instructions as it ran under SMT execution, and compute the single-threaded cycle component stacks following the single-threaded cycle accounting mechanism described in the previous chapter.

6.4.1 Cycle component prediction

Figure 6.8 compares the SMT cycle component stacks against the ST cycle component stacks for six example workloads assuming the ICOUNT fetch policy and dynamic resource partitioning (i.e., resources are fully shared between the threads, and resources are allocated on demand); these example workloads were chosen to represent an interesting sample of average workloads and a few extreme cases with the largest errors observed. Each graph shows the two ST cycle component stacks on the left and the two SMT cycle component stacks on the right;

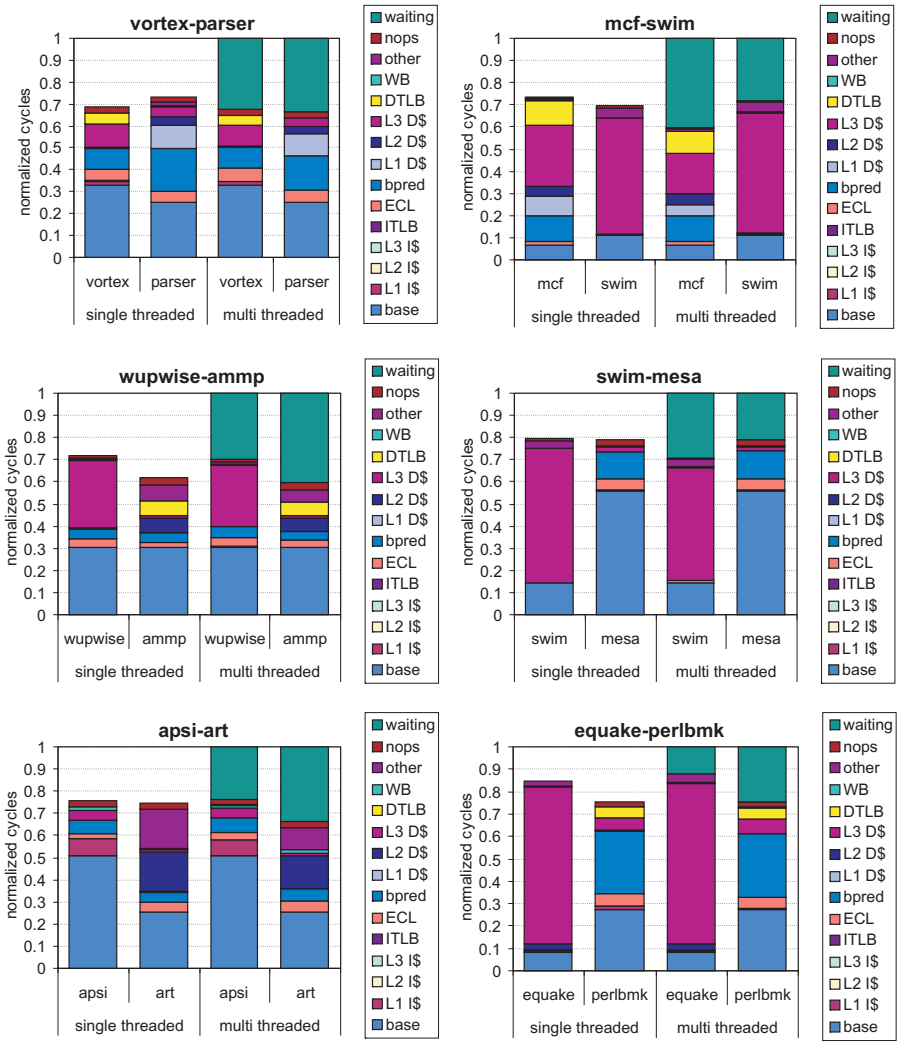

Figure 6.8: Evaluating SMT cycle component stacks versus single-threaded cycle component stacks for six example workloads assuming dynamic resource partitioning and the ICOUNT fetch policy.

the cycle component stacks are normalized to the SMT cycle component stacks. The workloads wupwise-ammp and equake-perlbmk show an almost perfectly predicted single-threaded cycle stack. The per-component prediction error is no larger than a few percent. The largest branch misprediction cycle component prediction error (6.1%) is observed for parser when co-executed with vortex. The reason for the inaccuracy is that the branch resolution time is underestimated: the critical path to the mispredicted branch tends to be smaller because there are fewer critical path instructions in the instruction window by the time the mispredicted branch is dispatched under SMT execution than under single-threaded execution. The largest 'other' cycle component error (10.7%) is observed for art when co-run with apsi. For the swim-mesa and mcf-swim workloads, the largest error appears for the long-latency (L3 D-cache) miss component for swim (7.6%) and mcf (12.8%, the largest error observed), respectively. From a detailed analysis we found the reason to be twofold: (i) the single-threaded MLP estimate tends to be an overestimation because it does not account for MLP limiters such as branch mispredictions between two long-latency loads that depend on the first long-latency load, and long-latency I-cache misses intervening two long-latency loads; and (ii) the SMT MLP calculation may be inaccurate because it also counts MLP along mispredicted paths.

6.4.2 Multiple fetch policies

To further assess the accuracy of the cycle accounting architecture across all workloads and multiple fetch policies, Figure 6.9 shows the average absolute cycle component prediction error, weighted by the contribution of this component to total execution time, for six resource sharing policies:

(i) Static partitioning (each thread can occupy at most half of the ROB, issue buffers and physical registers) and the round-robin fetch policy (instructions are fetched from another thread each cycle).

(ii) Static partitioning and the ICOUNT fetch policy [97] (fetch priority is given to the thread having the least instructions in the processor front-end pipeline and issue buffers).

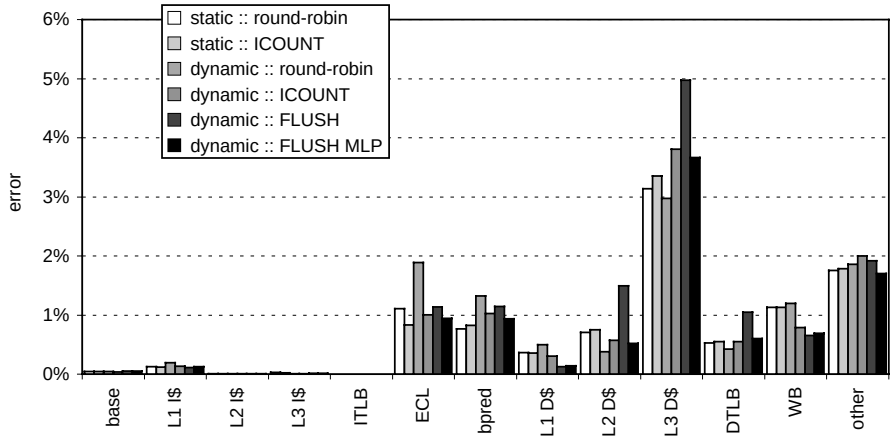

Figure 6.9: The average absolute cycle component prediction error across all workloads and six SMT processor fetch policies.

(iii) Dynamic partitioning (all resources are fully shared) and round-robin.

(iv) Dynamic partitioning and ICOUNT.

(v) Dynamic partitioning and flush [96] (when a thread encounters a long-latency load miss, all instructions of this thread are flushed, and the thread is fetch stalled until the data returns from memory).

(vi) Dynamic partitioning and MLP-aware flush [28] (when a thread encounters a long-latency load miss, its MLP potential is predicted and the instructions that do not contribute to MLP exploitation are flushed, see Appendix C).

Figure 6.9 shows that the cycle accounting architecture is accurate across multiple SMT fetch policies with average absolute cycle component prediction errors of a few percent. The largest error (no more than 5% on average) is observed for the long-latency load (L3) cycle component; as mentioned before, the reason is the difficulty in estimating single-threaded MLP as well as computing MLP under SMT execution.

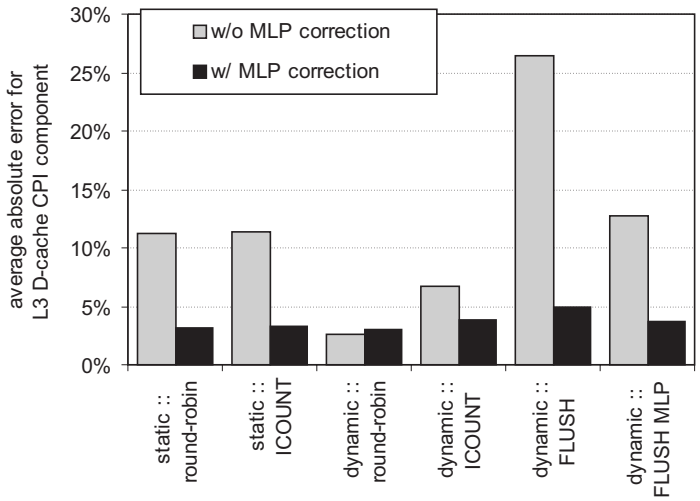

Figure 6.10: The average absolute L3 D-cache cycle component prediction error across all workloads for six fetch policies with and without MLP correction.

6.4.3 Importance of MLP correction

Figure 6.10 illustrates the importance of MLP correction for estimating the long-latency load (L3) cycle component: the average absolute prediction error is shown for the L3 D-cache cycle component for the six SMT fetch policies, both with and without MLP correction—the other cycle components are not affected by MLP correction. The average error drops below 5% across all fetch policies with MLP correction. Without MLP correction, the average error is typically higher and can be as high as 26.5% for the flush policy. The reason for the high error without MLP correction is the discrepancy in exploitable MLP under single-threaded execution versus SMT execution: a thread in single-threaded execution gets the entire ROB for exposing MLP, whereas a thread in an SMT processor typically does not. This is magnified in the flush policy, which flushes a thread that experiences a long-latency load, i.e., the flushed thread does not get to expose MLP, which explains the high error without MLP correction. The MLP correction mechanism corrects the observed MLP under SMT execution with the estimated MLP under single-threaded execution.

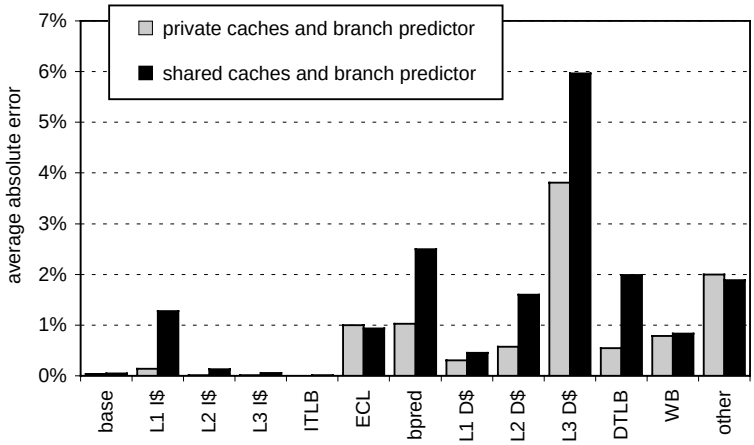

Figure 6.11: Average absolute cycle component prediction error assuming private versus shared caches and branch predictor, assuming dynamic resource partitioning and the ICOUNT fetch policy.

6.4.4 Private versus shared caches and branch predictor

In the results just presented, we assumed a partitioned memory hierarchy and branch predictor (performance isolation). A shared memory hierarchy and branch predictor (without performance isolation guarantees) may introduce additional conflict misses when co-executing threads compete for a shared resource. Figure 6.11 compares the average absolute error per cycle component compared to single-threaded execution assuming a partitioned versus a shared memory hierarchy and branch predictor. As expected, the errors tend to be higher assuming shared resources. The inaccuracy introduced through resource sharing could be addressed by estimating the number of additional conflict misses introduced through resource sharing as done by [12], or by guaranteeing a fair sharing mechanism as proposed by [34, 71, 69, 58, 49, 72, 50]. Although the error tends to be low for our setup, improving the accuracy of the cycle accounting architecture when applied to an SMT processor without performance isolation guarantees in the memory hierarchy and branch predictor is an interesting avenue for future work.

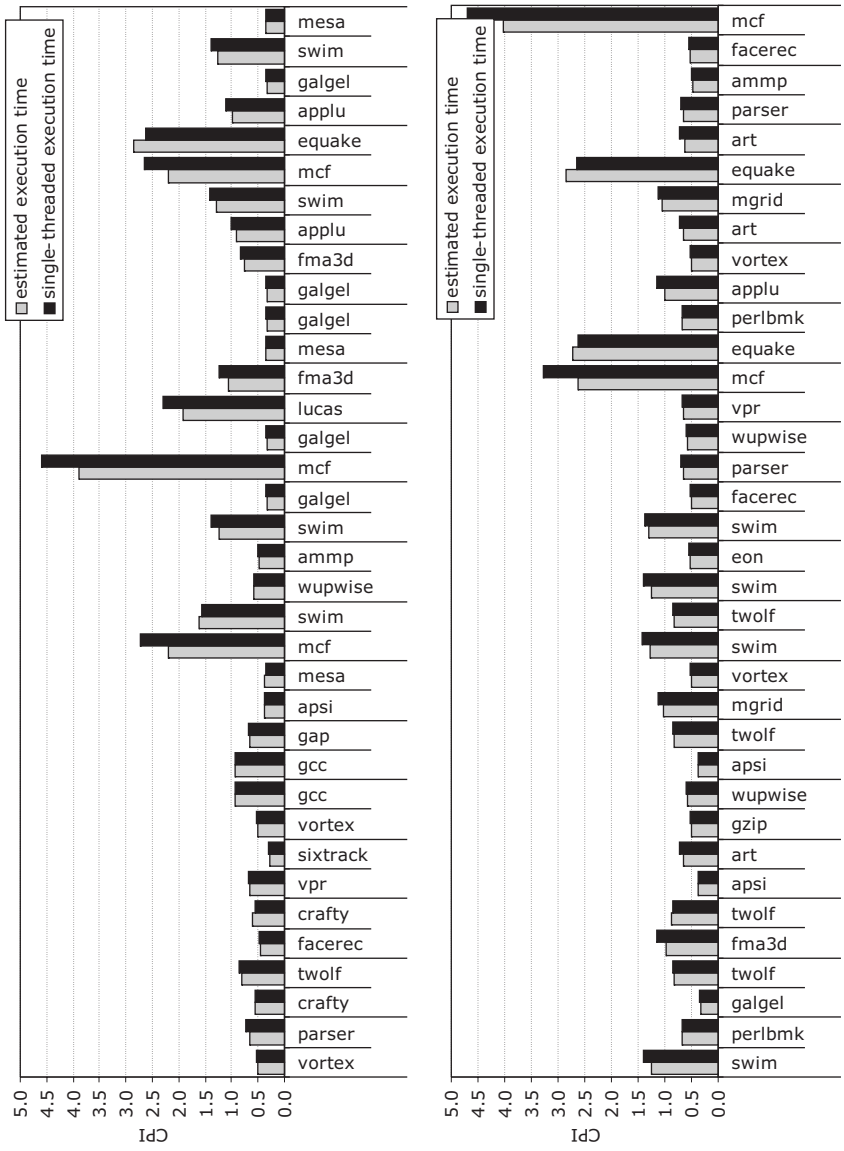

Figure 6.12: The estimated single-threaded virtual execution time (predicted through the cycle accounting architecture during SMT execution) versus the measured execution time under single-threaded execution, assuming dynamic resource partitioning and the ICOUNT fetch policy.

6.4.5 Per-thread progress prediction

So far, we were concerned with estimating cycle components. Adding the base cycle component with the miss event cycle components yields an estimation for the overall virtual cycle count under single-threaded execution, and dividing it by the number of instructions executed yields the estimated virtual single-threaded CPI. In Figure 6.12, the virtual single-threaded CPI prediction computed by the cycle accounting architecture during multi-threaded execution is compared against the CPI obtained from single-threaded execution, assuming ICOUNT and dynamic resource partitioning. These graphs show that the estimated execution time correlates well with the measured single-threaded execution time: the average absolute prediction error equals 6.8%. We observe similar results for the other resource sharing policies: static partitioning and round-robin (5.4%), static partitioning and ICOUNT (5.9%), dynamic partitioning and round-robin (6.5%), flush (7.2%) and MLP-aware flush (6.3%). These results demonstrate that the cycle accounting architecture can be used to assess per-thread progress. In particular, the cycle accounting architecture reports the number of cycles 'consumed' by each thread; this is the number of cycles the thread would execute in single-threaded mode, its virtual execution time. This is an abstraction consistent with the notion of timeslices in system software [91]: e.g., a thread consuming 75% of a timeslice's execution cycles for four timeslices on an SMT processor corresponds to consuming three timeslices on a single-threaded processor.

The cycle accounting method can thus be used by system software in order to have a more accurate view of the progress of each of the threads. Instead of assigning the complete CPU time to all of the concurrently executing threads, it can assign the correct single-threaded progress. With this information, system software may be able to provide a better quality of service to the SMT processor users, e.g., it can more accurately enforce priorities, and measure the number of cycles a thread has consumed, to provide a fair bill to users of a cycle-selling data center consisting of SMT machines.

6.5 Applications

The next three sections show how the cycle accounting performance counter architecture can be used to provide more quality of service on

an SMT processor. The next section shows how it can be used to monitor the performance of a running SMT machine. Section 6.5.2 uses the cycle accounting performance counters to drive a new SMT fetch policy that is able to provide more inter-thread fairness, and the last application is a fetch policy that enforces per-thread absolute or relative performance targets on an SMT processor. The applications discussed here are mainly hardware solutions, but as discussed earlier, the cycle accounting method can also be useful as an input to system software (operating systems and virtual machines) to improve quality of service guarantees to the users.

6.5.1 On-line SMT performance evaluation

For assessing the performance of an SMT processor running multi-program workloads, we need appropriate metrics. Appendix B discusses performance metrics for multi-program workloads, and concludes that metrics should have a system-level meaning. The proposed metrics are system throughput (STP, equivalent to the previously proposed weighted speedup metric [85]), and average normalized turnaround time (ANTT, equivalent to the reciprocal of the hmean metric [63]). We also consider the fairness metric proposed in [36], since it has some intuitive appeal.

$$STP = \sum_{i=1}^{n} \frac{IPC_i^{MT}}{IPC_i^{ST}} \tag{6.1}$$

$$ANTT = \frac{1}{n} \sum_{i=1}^{n} \frac{IPC_i^{ST}}{IPC_i^{MT}} \tag{6.2}$$

$$fairness = min_{i,j} \left(\frac{IPC_i^{MT}/IPC_i^{ST}}{IPC_j^{MT}/IPC_j^{ST}} \right) \tag{6.3}$$

with n the number of threads, IPC_i^{ST} the single-threaded IPC of thread i and IPC_i^{MT} the IPC of thread i in the multi-threaded execution. STP and fairness are higher-is-better metrics, while ANTT is a lower-is-better metric.

These metrics are all computed using the single-threaded IPC of the individual threads. This means that in order to quantify the performance of an SMT processor, a single-threaded execution or simulation has to be done. In a research and development context, this can be

justifiable, but to monitor the performance of an operational running machine, it is infeasible. Since the cycle accounting method described above provides single-threaded performance while running in multi-threaded mode, it can be used as an on-line monitoring mechanism. It can continuously and uninvasively report per-thread progress as well as overall system performance to the operating system and users. This information can be used to trigger compensations (e.g., changing the job schedule) when SMT performance is degrading.

Figure 6.13 shows the STP, ANTT and fairness values measured with the cycle accounting architecture compared to using post-execution single-threaded executions / simulations. For most of the workloads, the cycle accounting architecture is able to estimate performance metrics quite well, the average absolute error is 4.2% for the STP estimates, 4.6% for the ANTT estimates, and 7.9% for the fairness estimates. However, some show larger differences compared to the real values. The largest errors are observed in the fairness estimation, and are due to the fact that in some combinations, single-threaded IPC is underestimated for one thread and overestimated for the other (e.g., mcf-swim, mcf-equake, and equake-art; see also Figure 6.12). These effects strengthen each other, resulting in a worse estimate.

6.5.2 Thread-progress aware fetch policies

Optimizing SMT performance incurs a delicate balance between system throughput and per-thread performance, i.e., an SMT processor should be fair and should not starve threads while maximizing system throughput. The cycle accounting architecture presented in this chapter enables leveraging on-line fairness computation and per-thread progress indicators as the key enabler for a new class of *thread-progress aware* SMT fetch policies. For example, the goal of the fetch policy could be to have absolute fairness, i.e., the threads have equal single-threaded progress. This can be done by giving priority to threads that have made the least single-threaded progress. We propose three fetch policies along this line:

- *Waiting-Priority* gives higher fetch priority to a thread with a larger number of waiting cycles (and thus a lower number of base plus miss event cycles). By doing so, we strive at achieving equal progress for all threads.

Figure 6.13: STP (a), ANTT (b) and fairness (c) values, predicted by the cycle accounting architecture, compared to the real values (calculated using post-execution single-threaded simulations), assuming dynamic resource partitioning and the ICOUNT fetch policy.

- *Waiting-Priority-Stall* includes Waiting-Priority, and when the single-threaded V-ROB gets exhausted, the thread is stalled.
- *Waiting-Priority-Flush* includes Waiting-Priority-Stall, and when dispatch stalls, the thread with the largest single-threaded V-ROB is flushed for half the difference in V-ROB instructions with the smallest V-ROB thread.

The latter two policies strive at improving throughput while guaranteeing fairness; this is done by preventing a thread from clogging resources and, if it does clog resources, de-allocate part of the clogged resources.

Figure 6.14 compares these thread-progress aware fetch policies against the six other fetch policies considered in this chapter in terms of fairness, STP and ANTT; these are average numbers across all workloads. According to the results presented in Figure 6.14, dynamic partitioning in combination with round-robin performs worst in terms of fairness, STP and ANTT (which is a lower-is-better metric, in contrast with to fairness and STP). The reason is that a thread experiencing a long-latency load may clog all the resources thereby preventing the other thread from making forward progress. The ICOUNT mechanism strives at balancing the resources in the front-end pipeline and issue queue to both threads, and improves all metrics. Static partitioning has a slightly higher STP than ICOUNT and dynamic partitioning at the expense of being less fair; this is in line with the observation made by Raasch and Reinhardt [79]. The thread-progress aware fetch policies achieve the highest fairness, and the Waiting-Priority-Flush policy achieves an STP comparable to the best non thread-progress aware fetch policies, and the lowest ANTT of all policies.

6.5.3 Per-thread performance targets on an SMT processor

Leveraging a thread-progress aware fetch policy, we can control the progress of the threads co-executing on an SMT processor and isolate per-thread performance on an SMT processor. In other words, we can control the relative progress of the co-executing threads, e.g., one thread should make twice as much progress as another thread. Or, we can isolate the performance of one thread and achieve a given performance target irrespective of the other threads that happen to be running concurrently, while utilizing left-over instruction bandwidth to optimize system throughput.

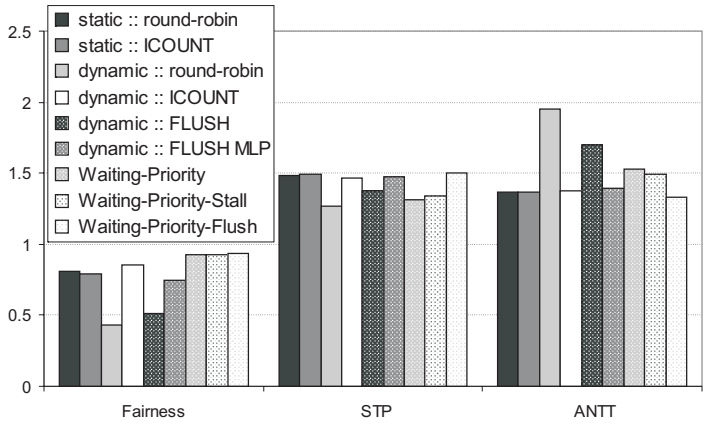

Figure 6.14: Comparing previously proposed fetch policies against the proposed thread-progress aware fetch policies in terms of fairness, STP and ANTT.

To this end, we introduce *Thread Progress Registers (TPRs)*, set by system software, that determine the fraction of the total execution cycles that should be devoted to each thread. The sum of all TPRs should not be larger than one, but may be lower. For example in a two-thread system, a TPR configuration of 75%–25% means that thread 1 and thread 2 should get at least 75% and 25% of the total execution cycles, respectively; the additional instruction bandwidth due to overlap effects during SMT execution can be distributed among both threads for improving system throughput. (A 50%–50% TPR configuration corresponds to a fair fetch policy, as described in the previous section.) A TPR configuration of 75%–NA (i.e., no TPR is set for thread 2) targets at thread 1 getting 75% of the total execution cycles, and the left-over instruction bandwidth is devoted to thread 2. Note that the meaning of the TPRs is consistent with the general understanding of timeslices in system software for managing thread priorities, QoS and performance isolation. For example, a 75%–25% TPR configuration could be realized in a single-threaded processor system through time multiplexing by giving 75% of the timeslices to thread 1 and 25% of the timeslices to thread 2. An SMT processor with a thread-progress aware fetch policy and TPRs can provide this same abstraction to system software while achieving significantly better system throughput.

A thread-progress aware fetch policy with TPRs operates as follows.

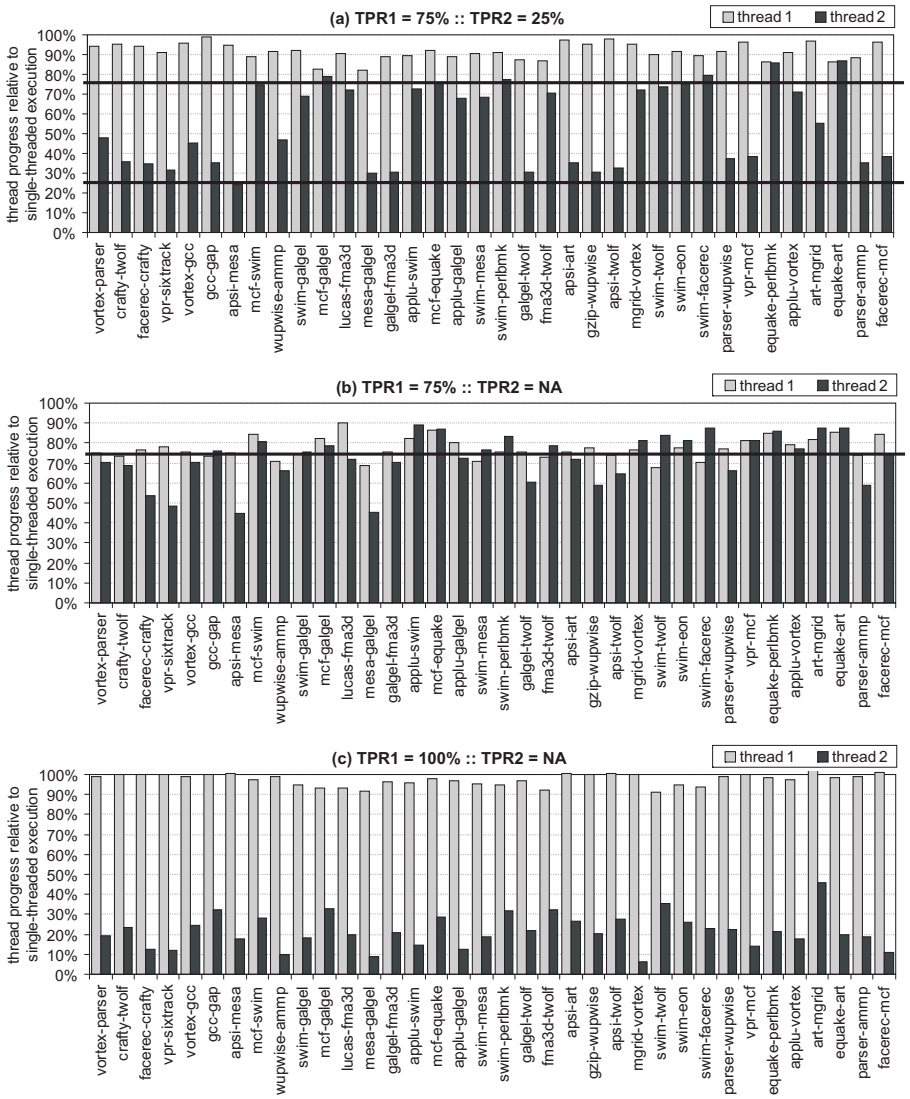

Figure 6.15: Evaluating SMT performance targets under three scenarios: (a) $TPR_1 = 75\%$ and $TPR_2 = 25\%$, (b) $TPR_1 = 75\%$ and $TPR_2 = NA$, and (c) $TPR_1 = 100\%$ and $TPR_2 = NA$.

For each thread i, the hardware thread priority controller computes the relative progress indicator P_i:

$$P_i = (C_{base,i} + C_{miss\ event,i}) - C_{total}\ TPR_i,$$

with C_{total} the total cycle count, TPR_i the thread's TPR, and $C_{base,i} + C_{miss\ event,i}$ the thread's base plus miss event cycle count (the thread's virtual execution time). The fetch policy then gives priority to the thread with the lowest P_i at run time, i.e., the thread that is lagging behind its target performance the most. A special case is included in the thread priority controller if one of the TPRs is not-applicable (NA): a thread with $TPR \neq NA$ is given priority as long as its P_i is negative; any left-over dispatch slots are given to the other threads with the NA TPR.

This policy for controlling per-thread performance is evaluated in Figure 6.15. We consider three scenarios on a two-thread SMT processor, assuming the Waiting-Priority-Flush policy: (a) $TPR_1 = 75\%$ and $TPR_2 = 25\%$; (b) $TPR_1 = 75\%$ and $TPR_2 = NA$; and (c) $TPR_1 = 100\%$ and $TPR_2 = NA$. For all (or most) of the threads, the fetch policy is able to reach per-thread performance close to or above its TPR: in case (a), the two threads achieve a performance above the 75% and 25% thresholds, respectively; in cases (b) and (c), most of the TPR_1 threads achieve performance above the 75% threshold and close to the 100% threshold, respectively. The left-over computation bandwidth above the TPR thresholds is distributed across both threads, see case (a), or is consumed by the other thread, see cases (b) and (c); the excess computation bandwidth improves system throughput on SMT processor not realizable through time multiplexing tasks on single-threaded processors.

The main cause of certain threads not reaching the TPR threshold in Figure 6.15(b) (only 3 threads out of the 36 two-thread combinations) is the delay of the MLP correcting mechanism. The single-threaded MLP is calculated at the commit stage when a number of instructions equal to the ROB size are committed after the load miss commits. This is some time after the occurrence of the long-latency load miss, and in the mean time, single-threaded execution time is overestimated, resulting in a higher P_i and thus a lower priority. In many cases (except for the $TPR = 100\%$ case), this is compensated by giving the thread a higher priority after the MLP correction, but in some cases, there is not enough time left to fully compensate for this MLP correction delay.

6.6 Related work

SMT processors. There is no prior work that proposes a per-thread cycle accounting architecture for SMT processors, however, several approaches have been proposed for managing QoS in SMT processors.

Cazorla et al. [9, 11] target QoS in SMT processors through resource allocation. They propose a system that samples single-threaded IPC, and dynamically adjusts the resources to achieve a pre-set percentage of the single-threaded IPC. The main difference with the cycle accounting architecture proposed in this chapter is that enforcing performance targets is one of the possible applications of the counter architecture. Cazorla et al.'s approach cannot compute per-thread virtual execution time as does our counter architecture. Regarding the QoS-aware fetch policies, there are three key differences between this work and ours: (i) Cazorla et al.'s approach requires a sampling phase (5% of the time) during which single-threaded IPC is measured while suspending co-executing threads whereas we compute single-threaded progress on the fly without run time overhead, (ii) Cazorla et al.'s approach incurs tuning overhead for finding the right resource allocation for achieving a pre-set target performance, and (iii) Cazorla et al.'s approach uses the single-threaded IPC of samples that are taken thousands of instructions earlier, while we estimate the instantaneous single-threaded IPC. The feedback loop in our approach is therefore much shorter than in the approach of Cazorla et al.

Cota-Robles [17] describes an SMT processor architecture that combines OS priorities with thread efficiency heuristics (outstanding instruction counts, number of outstanding branches, number of data cache misses) to provide a dynamic priority for each thread scheduled on the SMT processor. The key difference with our work is that the cycle accounting architecture proposed in this chapter enables a per-thread progress indicator as input to the fetch policy rather than a thread efficiency heuristic which is an indirect progress indicator.

Snavely et al. [86] study symbiotic job scheduling on SMT processors that strive at maximizing system throughput while satisfying job priorities. Jain et al. [51] study symbiotic job scheduling of soft real-time applications on SMT processors. Fedorova et al. [33] find that non-work-conserving scheduling, i.e., running fewer threads than the SMT processor allows, can improve system performance; they use a simple analytical model to find cases when a non-work-conserving policy is

beneficial.

SOE processors. Gabor et al. [36] propose fairness enforcement based on a cycle accounting mechanism for coarse-grain switch-on-event (SOE) multi-threaded processors. The cycle accounting architecture estimates single-thread performance for each of the threads had each of them been executed alone, while they are running in SOE multithreading. Our work concerns with cycle accounting for SMT processors rather than SOE processors, which poses a number of additional challenges because of the tighter interactions between co-executing threads on an SMT processor.

CMP platforms. A significant body of recent work has been done on QoS management in CMP-based computer systems. Unmanaged shared resources such as caches, off-chip bandwidth and the memory system, can affect per-thread and system performance in unpredictable ways and may starve threads. Various researchers have proposed software solutions [34] and hardware solutions for managing shared cache storage [49, 58], shared cache storage and memory bandwidth [50, 72], and CMP memory systems [69, 71].

6.7 Summary

This chapter presented a cycle accounting method for SMT processors that is able to estimate the single-threaded progress of each of the individual threads during multi-threaded execution. It is based on the insights provided by interval analysis and the single-threaded cycle accounting method of the previous chapter. The main extension needed was the ability to measure the impact of the interactions between the threads due to the sharing of resources. This has led to a new type of cycle component in addition to the base and the miss event penalty components, namely the 'waiting' component. This component reflects the amount of time a thread cannot make progress because of the fact that the processor is doing computation work for another thread.

The calculation of the impact of shared resources turned out to be relatively straightforward for front-end resource sharing (fetch and dispatch bandwidth), but is more complex for shared back-end resources (reorder buffer). In order to determine the per-thread penalty of back-

end miss events (which has to be equal to the penalty they would have observed when they were executed on a single-threaded processor), we need to keep track of the 'virtual ROB' (V-ROB) occupancy of each of the threads, i.e., the number of in-flight dispatch and waiting slots. The cycle accounting architecture then counts back-end miss event penalties as soon as the V-ROB occupancy equals the actual ROB size. We also have to compensate for the loss of memory-level parallelism due to the fact that threads usually cannot occupy all ROB entries to expose the maximum MLP.

The end result is a cycle accounting architecture that is able to reconstruct single-threaded cycle component stacks for each of the individual threads that are executing concurrently on an SMT processor. By subtracting the waiting component of the total cycle count, the single-threaded progress (its virtual single-threaded execution time) of the co-executing threads can be estimated, which provides valuable information to enforce quality of service for SMT processors. We showed how it can be used to do on-line performance monitoring of SMT processors, to enable thread-progress aware fetch policies, and to enforce performance targets on SMT processors.

Chapter 7

Conclusions and future work

The real danger is not that computers will begin to think like men,
but that men will begin to think like computers.
Sydney J. Harris

7.1 Summary

Performance analysis of contemporary complex processor architectures is a challenging task. Performance improving techniques such as pipelining, superscalarity, out-of-order execution, speculative execution and caching have boosted processor performance, but make it more difficult to attain intuitive insight into their performance impact. Therefore, performance evaluation in an experimental context has shifted towards simulation. Simulation provides accurate performance results, but is highly time-consuming and also gives no fundamental insight into the performance of a program running on a processor. This lack of insight is also reflected in on-line performance monitoring tools. Existing hardware performance counters can measure various events that have an impact on processor performance, but they fail to accurately quantify this impact.

In this work we showed that by studying the dispatch behavior of a program executing on a processor, a sufficiently detailed but intuitive performance model can be deduced, called *interval analysis*. This model reveals some interesting insights:

- In the absence of miss events, a balanced processor achieves a performance level equal to the designed processor (dispatch) width.

- The performance impact of an instruction cache miss equals the time needed to access the appropriate cache level or memory. The penalty for an I-TLB miss equals the time needed to consult the page table in memory.
- The penalty of a branch mispredictions consists of the branch resolution time, i.e., the time needed to execute the instructions the branch depends on, plus the front-end refill time. Due to the branch resolution time, the branch misprediction penalty can be much larger than the front-end pipeline depth. The branch resolution time depends on the interval length distribution (i.e., the distribution of the number of instruction between the miss events), the available ILP and the average instruction execution latency. In balanced processors, branch mispredictions are the only case where the critical dependency path of an application affects performance.
- Short back-end misses (L1 data cache misses) and medium-latency functional units (multiply, divide) usually show no or little performance impact, because they are hidden through out-of-order execution in a balanced processor design.
- Long-latency back-end misses incur a penalty because they cause the exhaustion of instruction window entries. Their penalty can be approximated by the memory access time, but overlaps between independent long-latency load misses (memory-level parallelism or MLP) can substantially lower the individual penalty of long-latency load misses.

Using the insights provided by interval analysis, we developed a mechanistic performance model, that approximates total performance without doing extensive simulations, and that provides more insight into the performance impact of miss events (cache misses and branch mispredictions). This is done by estimating the execution time in the absence of miss events and by approximating the penalties of each of the miss events. The input for the model are application-dependent characteristics (such as average critical dependency path length and instruction mix), processor parameters (e.g., dispatch width and instruction execution latencies), and program locality behavior (branch predictor accuracy and cache miss rates), which depends on both the application and the processor configuration. The model has a performance estimation error of 6.9% for a four-wide baseline processor. Ap-

plying the model to a study of the processor pipeline depth and width has yielded an interesting relationship between optimal pipeline depth and width, i.e., if the pipeline width increases by a factor c, the optimal pipeline depth decreases with a factor $\sqrt{c}$.

Interval analysis also provides the needed insights needed to design a hardware performance counter architecture that quantifies the impact of miss events by decomposing total execution time into base and miss event components. We have shown that this can be done in a limited chip area budget, and that it yields far more accurate cycle component stacks than previously proposed cycle accounting methods. The usefulness of accurate cycle component stacks for software designers (programmers and compiler builders) is illustrated by a study of the performance impact of compiler optimizations. This study confirms and quantifies some well-known facts, but also reveals some interesting insights, such as the fact that instruction scheduling is less important and can be even harmful for out-of-order processors compared to in-order processors, while avoiding miss events and exploiting MLP has more performance impact on out-of-order processors.

Finally, interval analysis also provides insight into how to do accurate cycle accounting in SMT processors. The main problem in multi-threaded processors is the mutual performance impact of the threads, which makes it difficult to isolate the performance of the individual threads while they are co-running on an SMT processor. Therefore, we developed a counter architecture that estimates the single-threaded progress of the individual threads. It reconstructs the cycle component stacks of each of the threads as if they were executing on a single-threaded processor. Being able to estimate single-threaded progress during multi-threaded execution yields a far more accurate view of the actual progress of each of the individual threads, which is useful for system software or hardware to provide quality of service on SMT processors. Using this performance counter architecture, we developed a fetch policy that is able to enforce performance targets for individual threads, irrespectively of the co-executing threads.

7.2 Future work

In this thesis, we developed performance estimation and analysis tools that illustrate both the simplicity and power of interval analysis. How-

ever, more research can be conducted to fully expand the capabilities of the analytical performance model.

Improving the applicability of the mechanistic model

The mechanistic model does not model some mechanisms present in contemporary processors, such as hardware prefetching. It should be studied how to model the performance impact of these mechanisms and how to integrate them into the existing model.

The mechanistic model can also be used to evaluate the performance impact of new microarchitectural techniques, without the need for detailed simulations. Using interval analysis, the performance impact of these new techniques can be modeled and integrated into the overall model. For example, the performance impact of runahead execution [70] can be assessed by estimating the gain in MLP it provides, and estimating the penalty of re-executing some instructions.

Improving the accuracy of the SMT cycle accounting architecture

The main limitation of the proposed cycle accounting architecture for SMT processors is that it is less accurate when we assume shared branch predictors and caches. Augmenting the counter architecture with an estimator of the number of additional conflict misses through sharing will improve its accuracy when no fair sharing is guaranteed.

Dynamic performance optimization using the cycle accounting architectures

Both the cycle accounting architectures for single-threaded processors and SMT processors provide valuable performance-related information while the application(s) is (are) running. This information can be used by system software or dynamic compilers to perform at runtime performance optimization. Using the on-line measured cycle component stacks, a dynamic compiler can detect the main performance inhibitors of a frequently executed method, and apply specific code optimizations directed at reducing the largest cycle components. The cycle accounting architectures may also provide valuable information for operating systems and (system) virtual machines to support the distribution of the available resources among the tasks, in order to maximize system

throughput and / or to minimize the turnaround time of the individual tasks.

Analytical performance models for multi-threaded and multi-core processors

Interval analysis as elaborated in Chapter 3 is an intuitive way to analyze the performance of single-threaded processors. As shown in Chapter 6, its base mechanisms can also be used to study multi-threaded processor performance. Currently, we only used it to develop a cycle accounting architecture for SMT processors, but it would be interesting to use it as a basis to develop a performance estimation model for multi-threaded processors (SMT and switch-on-event), as we have done in Chapter 4 for single-threaded processors.

The mechanistic performance model for single-threaded processors can also be integrated in an analytical performance model for chip multiprocessors (CMPs). Since it already provides an abstraction of the execution of the individual instructions on a single processor core, it can be used to model the performance of each of the processor cores the CMP consists of. By developing models to characterize cache sharing, cache coherence traffic, shared off-chip bandwith, etc., the overall performance of the CMP can be modeled and estimated.

Appendix A

Simulation details

A.1 Single-threaded processor

A.1.1 Simulator

For all single-threaded processor simulations we used SimpleScalar v3.0 with the Alpha instruction set [7].

A.1.2 Benchmarks

We use the SPEC CPU 2000 benchmark suite (see www.spec.org). The binaries of these were taken from the SimpleScalar website. Unless mentioned otherwise, a single 100 million instruction SimPoint [82, 78] is used to do the evaluations. The benchmarks, along with their inputs and SimPoints are represented in Table A.1.

A.1.3 Processor configuration

Unless mentioned otherwise, the baseline processor configuration is that presented in Table A.2.

For the processor depth and width studies in Section 4.4, we changed some of the processor parameters as function of the depth and width, see Table A.3 and Table A.4, respectively.

Table A.1: Benchmarks, inputs and SimPoints (in billions of instructions skipped) for single-threaded simulations.

Benchmark	input	SimPoint
bzip2	program	0.9B
crafty	ref	0B
eon	rushmeier	1.8B
gap	ref	209.4B
gcc	166	9.9B
gzip	graphic	0.3B
mcf	ref	31.6B
parser	ref	1.6B
perlbmk	makerand	1B
twolf	ref	3.1B
vortex	ref2	5.9B
vpr	route	7.1B
ammp	ref	212.9B
applu	ref	1.7B
apsi	ref	4.5B
art	ref-110	6.6B
equake	ref	19.3B
facerec	ref	13.5B
fma3d	ref	29.7B
galgel	ref	314.9B
lucas	ref	3.4B
mesa	ref	8.8B
mgrid	ref	0.5B
sixtrack	ref	8.1B
swim	ref	0.4B
wupwise	ref	58.3B

Table A.2: Baseline processor configuration.

parameter	value
ROB	128 entries
LSQ	64 entries
processor width	decode, dispatch, issue and commit 4 wide fetch 8 wide
fetch buffer	8 entries
latencies	load (2), mul (3), div (20)
L1 I-cache L1 D-cache L2 cache main memory	8KB direct-mapped 16KB 4-way set-assoc unified, 1MB 8-way set-assoc, 9 cycles 250 cycle access time
branch predictor front-end pipeline	hybrid bimodal / gshare predictor 5 stages

Table A.3: Processor resource scaling as a function of pipeline depth.

Parameter	depth 1x	depth 2x	depth 3x	depth 4x	depth 5x
ROB	128	256	384	512	640
LSQ	64	128	192	256	320
latencies:					
integer	1	2	3	4	5
load	2	4	6	8	10
mul	3	6	9	12	15
div	20	40	60	80	100
L2 cache access time	9	18	27	36	45
main memory access time	250	500	750	1000	1250
front-end pipeline depth	5	10	15	20	25

Table A.4: Processor resource scaling as a function of processor width.

Parameter	width 2	width 4	width 6	width 8
ROB	64	128	256	512
LSQ	32	64	128	256
fetch width and fetch buffer size	4	8	12	16
dispatch, issue and commit width	2	4	6	8

A.2 SMT processor

A.2.1 Simulator

We used the SMTSIM simulator version 2.0 alpha [95] to do the SMT processor simulations. We added a write buffer to the simulator's processor model: store operations leave the reorder buffer upon commit and wait in the write buffer for writing to the memory subsystem; commit blocks in case the write buffer is full and we want to commit a store.

A.2.2 Benchmarks

We use the SPEC CPU2000 benchmarks with reference inputs, see Table A.5. These benchmarks are compiled for the Alpha ISA using the Compaq C compiler (`cc`) version V6.3-025 with the `-O4` optimization option. For all of these benchmarks we select 200M instruction (early) simulation points using the SimPoint tool [82, 78].

A.2.3 Processor configuration

Unless mentioned otherwise, the baseline processor configuration is that presented in Table A.6.

Table A.5: Benchmarks, inputs and SimPoints (in billions of instructions skipped) for multi-threaded simulations.

Benchmark	input	SimPoint
bzip2	graphic	4B
crafty	ref	2.2B
eon	chair	11.6B
gap	ref	24.6B
gcc	166	0B
gzip	program	27.2B
mcf	ref	14B
parser	ref	11.8B
perlbmk	makerand	0.4B
twolf	ref	42B
vortex	ref1	13.4B
vpr	place	33.4B
ammp	ref	2.8B
applu	ref	1.6B
apsi	ref	5B
art	ref 110	0.4B
equake	ref	21.4B
facerec	ref	0B
fma3d	ref	24.8B
galgel	ref	4B
lucas	ref	3.4B
mesa	ref	2.6B
mgrid	ref	18.4B
sixtrack	ref	8.4B
swim	ref	1B
wupwise	ref	4.4B

Table A.6: The baseline SMT processor configuration.

parameter	value
fetch policy	ICOUNT 2.4
(shared) reorder buffer size	256 entries
instruction queues	96 entries in both IQ and FQ
rename registers	200 integer and 200 floating-point
processor width	4 instructions per cycle
functional units	4 int ALUs, 2 ld / st units and 2 FP units
front-end pipeline	11 cycles
branch predictor	2K-entry gshare
branch target buffer	256 entries, 4-way set associative
write buffer	24 entries
L1 instruction cache	64KB, 2-way, 64-byte lines
L1 data cache	64KB, 2-way, 64-byte lines
unified L2 cache	512KB, 8-way, 64-byte lines
unified L3 cache	4MB, 16-way, 64-byte lines
instruction / data TLB	128 / 512 entries, fully-assoc, 8KB pages
cache hierarchy latencies	L2 (11), L3 (35), MEM (350)

Appendix B

Multi-program workload performance metrics

Performance metrics are at the foundation of experimental computer science and engineering. Researchers and engineers use quantitative metrics for assessing the performance of their novel research ideas and engineering progress: needless to say that adequate metrics are of primary importance, whereas inappropriate metrics may drive research and development in a wrong or unfruitful direction.

For single-threaded processors, the sole performance metric is the execution time of a program. The faster a processor can execute a program, the better is its performance. Performance is often reported in terms of IPC, instructions executed per cycle, or in its reciprocal form as CPI (cycles per instruction). IPC can be easily measured by dividing the number of instructions executed by the number of cycles needed to execute these instructions.

IPC values are application-dependent. A program that accesses a big chunk of memory in an unpredictable way has more cache misses and will have a lower IPC than a program that accesses less data and/or has a more regular memory access pattern. Therefore IPC should always be reported together with the application used to measure it. To compare processors, people have developed benchmark applications, e.g., the SPEC benchmark suite [45], the MiBench suite [41], and many others.

IPC can also be used in the context of multi-programmed workloads, i.e., multiple programs executing on a multi-threaded processor

(e.g., an SMT processor or switch-on-event processor). It is calculated by adding up the number of instructions executed for all threads, and dividing this by the number of cycles needed to execute these instructions. This is typically referred to as IPC throughput: the average number of instructions that the processor can execute per cycle, irrespective of the performance of the individual threads. This is a direct translation of the single-threaded IPC and was commonly used in the past, but it has some main drawbacks.

The first obvious drawback is that it does not take into account the performance degradation of the individual threads. A high throughput value does not mean that the performance of each of the individual threads is maximized. Optimizing IPC throughput can cause processors to prefer fast executing threads (with a high single-threaded IPC) above slower threads. This results in an unfair schedule.

A second, less obvious, drawback is that IPC throughput does not measure the efficiency of multi-threading. Since single-threaded IPC is application dependent, summing up these IPC values yields a performance metric that is also very application dependent: co-executing two slow threads will always give lower IPC throughput values than co-executing two fast threads, irrespectively of how efficient the threads are co-executed. Instead of considering the number of instructions executed per cycle, the efficiency of multi-threaded processors should be expressed in terms of 'threads per cycle': how many progress do each of the concurrently executing threads make, irrespective of the single-threaded IPC of the threads. This definition is equivalent to system throughput in a system software context, i.e., how many jobs are finished per time unit.

To calculate system throughput (STP), consider a situation where n threads are executing simultaneously on a multi-threaded processor, for a timeslice of C^{MT} cycles. During this timeslice, each thread has executed I_i instructions ($i = 1, 2, \ldots, n$). To execute the same number of instructions on a single-threaded processor, each thread would have needed C_i^{ST} cycles. So, during C^{MT} cycles, the total single-threaded progress of all threads equals $\sum_{i=1}^{n} C_i^{ST}$. System throughput can thus be calculated as:

$$STP = \frac{\sum_{i=1}^{n} C_i^{ST}}{C^{MT}} \tag{B.1}$$

If system throughput is equal to one, then the multi-threaded processor's performance is equal to time sharing on a single-threaded pro-

cessor: each thread gets a delimited time slice of total time, and all slices sum up to the total time. This means multi-threading adds no efficiency to the processor. If it is larger than one, multi-threading outperforms time sharing. The maximum value is n, indicating that all threads run at their full single-threaded speed, meaning that the n-context multi-threading processor performs as good as n single-threaded processors running concurrently. Theoretically, the value of system throughput is between 1 and n, but sharing branch predictors and caches can introduce extra conflict misses compared to single-threaded execution, which can cause the system throughput value to be less than one.

Since performance metrics for multi-threaded processors are often expressed in terms of the IPCs of the individual threads, we convert the above formula to IPC values instead of cycles.

$$STP = \sum_{i=1}^{n} \frac{C_i^{ST}}{C^{MT}} = \sum_{i=1}^{n} \frac{I_i/C^{MT}}{I_i/C_i^{ST}} = \sum_{i=1}^{n} \frac{IPC_i^{MT}}{IPC_i^{ST}} \tag{B.2}$$

This yields an expression that is equal to the weighted speedup metric, proposed in [85]. While it was introduced as a metric that has intuitive appeal, but with no founded system level meaning, we now found that it expresses system level throughput.

System throughput however still does not take into account explicitly the performance degradation of the individual threads. A multi-threaded execution where one thread yields 0.9 of its single-threaded execution performance and the other one 0.5 is less fair than when both threads show a performance degradation of 0.7, although they have the same system throughput value of 1.4. A metric that is more meaningful for end users is average normalized turnaround time (ANTT) (which is a lower-is-better metric, in contrast to system throughput):

$$ANTT = \frac{1}{n} \sum_{i=1}^{n} \frac{C^{MT}}{C_i^{ST}} = \frac{1}{n} \sum_{i=1}^{n} \frac{IPC_i^{ST}}{IPC_i^{MT}} \tag{B.3}$$

ANTT reflects the average degradation of the turnaround time of a thread (i.e., the time needed to complete a job) compared to its single-threaded turnaround time. Applied to the example above, the ANTT of the first example is 1.6, and 1.4 for the second example.

STP and ANTT are hence performance metrics for multi-threaded processors that have a system-level meaning. Though they are very closely related, STP reflects the efficiency of a multi-threaded processors, which is a system-oriented point of view, whereas ANTT reflects a

certain quality of service given to the end user. Balancing these metrics is more an economic decision than a pure performance related matter. For example, a data center—consisting of multi-threaded processors—that 'sells' computing cycles to their users can make more profit by optimizing system throughput (since more 'single-threaded cycles' are executed during one 'multi-threaded cycle'), or provide a better service to costumers by minimizing the average turnaround time degradation.

Two other metrics have been proposed in literature. One of them is fairness [36], which is defined as:

$$fairness = min_{i,j}\left(\frac{IPC_i^{MT}/IPC_i^{ST}}{IPC_j^{MT}/IPC_j^{ST}}\right) \tag{B.4}$$

$$= min_{i,j}\left(\frac{C_i^{ST}/C^{MT}}{C_j^{ST}/C^{MT}}\right) = min_{i,j}\left(\frac{C_i^{ST}}{C_j^{ST}}\right) \tag{B.5}$$

The goal of this metric is clear: optimizing it will strive at equal performance degradation for each thread. It is used as a counterbalance for throughput to ensure every thread has a fair share of the overall performance.

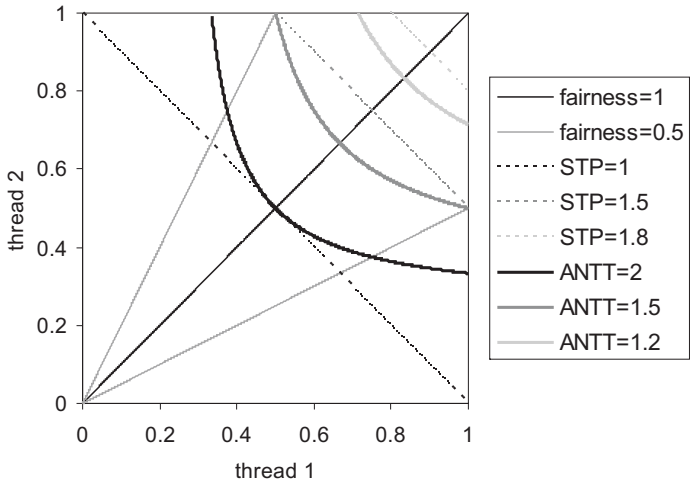

Figure B.1: Comparing STP, ANTT and fairness for a two-thread workload.

Figure B.1 graphically compares the STP, ANTT and fairness metrics for a two-thread workload. The axes show the relative progress

of each of the threads (i.e., C_i^{ST}/C^{MT} or IPC_i^{MT}/IPC_i^{ST}), and the lines show the points where a certain metric equals a certain value (see legend). The upper-right corner is the optimal point: both threads progress at their single-threaded speed. At that point fairness is 1, STP is 2 and ANTT equals 1. This graph shows that system throughput and fairness are almost independent of each other. Together, they reflect two base requirements of a multi-threaded processor, i.e., a multi-threaded processor should have good throughput and should be fair. ANTT on the contrary is a combined metric, balancing throughput and fairness, which can be explained by its user-oriented nature: it minimizes turnaround time (which is throughput-related) for all threads (which includes a notion of fairness).

Another widely used metric is hmean [63]. It resulted from the insight that weighted speedup does not harm unfair schedules. So, instead of summing all the speedup terms as done for weighted speedup, a harmonic mean of the speedups is taken, because the harmonic mean tends to stress lower values. When one thread has low performance and the other threads have higher performance, the harmonic mean will give more weight to the low value, penalizing unfair schedules. Hmean is calculated as

$$hmean = \frac{n}{\sum_{i=1}^{n} IPC_i^{ST}/IPC_i^{MT}} = \frac{n}{\sum_{i=1}^{n} C^{MT}/C_i^{ST}} \qquad \text{(B.6)}$$

Following this definition, hmean is the reciprocal of ANTT. As such, although hmean was designed as an artificial metric with no physical meaning, it is in fact the reciprocal of the average turnaround time degradation (which has a system level meaning). Hmean can therefore be used as a higher-is-better metric that corresponds to ANTT, but relative gains in hmean are meaningless. Improvements should always be reported in terms of ANTT reduction.

Appendix C

SMT fetch policies

Balancing throughput and fairness in an SMT context is done by implementing an efficient fetch policy. Each cycle the policy decides from which thread(s) instructions should be fetched in order to maximize throughput and not favor a single thread too much. Ever since the introduction of SMT in computer architecture research, people have acknowledged the importance of a fetch policy and many researchers have tried to optimize it [8, 10, 24, 96, 97]. Section C.1 discusses round-robin and ICOUNT, which are base fetch policies that have been proposed at the introduction of SMT. Section C.2 discusses fetch policies that improve the base policies in the case of a thread experiencing a long-latency load miss. Section C.3 then describes our proposed memory-level parallelism aware fetch policy. Finally, Section C.4 discusses some other proposed fetch policies and resource partitioning mechanisms.

C.1 Base fetch policies

A simple policy that at first sight seems to be fair by construction is round-robin [97]. That is, the fetch engine fetches instructions from a new thread each cycle, and when all threads have had their turn, it restarts from the first one. The number of instructions injected in the processor is equally distributed among all threads, which seems a fair starting point. But threads behave differently depending on inter-instruction dependencies, instruction execution latencies and the number of miss events they cause. A slow thread (with many miss events)

can execute less instructions per cycle than a fast thread, so when an equal number of instructions is fetched, the instruction window will fill up with instructions from the slow thread, while the fast thread consumes its instructions faster. This leaves increasingly less resources for the fast thread, which results in an unbalanced situation.

Therefore, Tullsen et al. [97] proposed a fetch policy named ICOUNT, that takes into account the number of in-flight instructions of each thread. A per-thread counter is incremented when an instruction of that thread is fetched, and decremented when an instruction is issued. By doing so, it contains the number of instructions of that thread in the front-end pipeline and the issue buffer. A thread with the lowest counter value is given higher priority to fetch instructions. This balances the front-end resource utilization of each thread, giving more fetch priority to fast threads. This is in line with the multi-threaded performance metrics: a fast thread should get more instructions not to degrade its performance too much relative to single-threaded performance, while slow threads, that already have a low single-threaded performance, are penalized less when fewer instructions are fetched.

C.2 Long-latency load aware fetch policies

There is however one case where ICOUNT falls short. When one thread encounters a long-latency load miss (that has to access main memory, or a TLB miss that has to consult the page table in memory), the reorder buffer will fill up with instructions of that thread because the load miss cannot commit. When the ROB fills up, the issue buffer will also fill with instructions dependent on the load miss, and ICOUNT will give this thread a low fetch priority. This is good, because the thread cannot make any progress due to the load miss, so it is useless to fetch more instructions for that thread. But since the other thread(s) can encounter instruction cache misses or taken branches that prevent a thread from using the full fetch width, there are still cycles where the long-latency thread is the only thread left from which instructions can be fetched, and the ROB will continue to fill with instructions from that thread. This eventually leads to a situation where one thread occupies most of the ROB entries without making progress, causing the other threads to slow down.

To overcome this problem, Tullsen and Brown [96] propose to take extra measures when a long-latency load miss is detected. They pro-

pose to stall the fetching of instructions of a thread experiencing a long-latency load miss. The thread remains stalled as long as the load miss is waiting for data. Furthermore, instructions already fetched after the load miss can be flushed (which can be a lot, since it takes a while to detect a long-latency load miss: first all instructions it depends on have to execute (load resolution time) and then all intermediate cache levels have to be consulted). The flushing of instructions frees even more resources for the other threads and therefore performs better, but it needs extra hardware for taking checkpoints to restore architectural state when the load miss returns. Cazorla et al. [8] try to avoid flushing by predicting long-latency load misses at the fetch stage and fetch stall threads when a long-latency miss is predicted. This avoids fetching instructions after the long-latency load miss, and makes a flushing mechanism unnecessary.

C.3 MLP-aware fetch policy

The long-latency load aware fetch policies discussed in the previous section, however, do not take into account the importance of memory-level parallelism. When a thread showing a long-latency load miss is fetch stalled and instructions following the load miss are flushed, the possible MLP cannot be exploited. This delays the long-latency load misses that could have been handled in parallel until after the load miss returns, which introduces substantial performance degradation compared to single-threaded execution. Threads having a lot of MLP potential are therefore severely delayed compared to single-threaded execution, worsening SMT performance.

Therefore, we develop an MLP-aware fetch policy [28] that balances the freeing of resources and the preservation of MLP in case a long-latency load miss is detected. To this purpose, we design an MLP predictor, presented in the next section. Section C.3.2 deals with the use of this predictor in an MLP-aware fetch policy.

C.3.1 MLP predictor

The purpose of an MLP predictor is to predict whether a certain long-latency load miss will expose MLP, i.e., if it is followed by other independent long-latency load misses within the instruction window size. To ease the design of the fetch policy, we decided not to predict the

actual MLP, i.e., the number of parallel independent long latency load misses, but the MLP distance, which is defined as the number of instructions that need to be fetched after the current load miss, in order to exploit the maximum MLP within the window size.

The predictor is a simple last value predictor, indexed by the instruction address of the load miss, which means that the predicted MLP distance is equal to the previously observed MLP distance of the load miss. The difficult part is how to update the predictor. In order to detect the maximum MLP of a thread, the ROB needs to be filled with instructions of that thread, which has an important performance impact for the other threads. Moreover, when the MLP of a certain static load enlarges, e.g., when it first showed no MLP and later on exposes some MLP, the maximum MLP cannot be detected, since the predictor steers the fetch policy by stalling the fetch of new instructions after the predicted distance.

The solution is to use a shift register per thread that contains the long-latency load miss history of the most recent instructions. A zero is shifted in if the last instruction was not a long-latency load miss, and a one if it was. If a one hits the head of the shift register, then the MLP distance can be obtained by looking at the position of the most recently inserted one. By also retaining the instruction addresses of the load misses, the predictor table can be indexed and updated.

We decided to put the shift registers at the commit stage for three reasons. First, at commit, the instruction is finished and it is known if it caused a long-latency load miss or not, which is not the case at for example dispatch. Second, instructions along mispredicted paths never commit, so MLP along mispredicted paths is not calculated (which should not, since at the next occurrence of this load miss, the branch may be correctly predicted). And third, since the MLP distance will be needed by the fetch engine to decide how many instructions should be fetched, it has to reflect the number of instructions between load misses as they appear in program order. Since commit handles instructions in-order, the number of instructions committed between two load misses is equal to their mutual distance in program order (in contrast to for example the issue stage that is out-of-order: the number of instructions issued between the issuing of two load misses is usually not equal to their mutual distance in the dynamic instruction trace).

C.3.2 Fetch policy

The fetch policy is now very intuitive: if a long-latency load is detected, the MLP distance is predicted and the number of instructions already fetched after the load miss is determined. If this number of instructions is larger than the MLP distance, then a number of instructions have to be flushed such that the remaining number of instructions equals the MLP distance. If too few instructions are fetched, fetching should continue until the predicted MLP distance is reached. If the appropriate number of instructions are flushed or fetched, the thread is fetch stalled (i.e., no instructions of that thread are fetched anymore) until the load miss returns. The MLP predictor can also be used in combination with a load miss predictor, avoiding the need of a flush mechanism.

Since an MLP-intensive thread still holds a number of resources (ROB entries) for a long time, the predicted MLP distance should be limited to ensure a fair share of the resources for the other threads. We decided to limit the MLP distance to the total ROB size divided by the number of threads. This can be done by limiting the size of the shift registers. We also evaluated the policy with smaller and larger shift registers, but dividing the ROB size equally among the threads turned out to be optimal.

C.4 Other fetch policies and resource partitioning mechanisms

Besides the fetch policies discussed earlier in this appendix, other fetch policies have been proposed as well. However these fetch policies do not address the long-latency load problem and are orthogonal to our MLP-aware fetch policy. For example, El-Moursy and Albonesi [24] propose to give fewer resources to threads that experience many data cache misses. They propose two schemes, namely data miss gating (DG) and predictive data miss gating (PDG). DG drives the fetching based on the number of observed L1 data cache misses, i.e., by counting the number of L1 data cache misses in the execute stage of the pipeline. The more L1 data cache misses observed, the fewer resources the thread can allocate. PDG strives at overcoming the delay between observing the L1 data cache miss and the actual fetch gating in the DG scheme by predicting L1 data cache misses in the frontend pipeline stages. Another scheme by Cazorla et al. [10] proposes to monitor the dynamic

usage of resources by each thread and strives at giving a fair share of the available resources to all the threads. The input to their scheme consists of various usage counters for the number of instructions in the instruction queues, the number of allocated physical registers and the number of observed L1 data cache misses. Choi and Yeung [13] go one step further and use a learning-based resource partitioning policy.

Bibliography

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, pages 248–259, June 2000.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of the 25th Very Large Database Conference*, 1999.

[3] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, November 1997.

[4] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, February 2002.

[5] D. Boggs, A Baktha, J. Hawkins, D.T. Marr, J.A. Miller, P. Roussel, R. Singhal, B. Toll, and K.S. Venkatraman. The microarchitecture of the Intel Pentium 4 processor on 90nm technology. *Intel Technology Journal*, 8(1):1–17, 2004.

[6] P. Bose and T. M. Conte. Performance analysis and its impact on design. *IEEE Computer*, 31(5):41–49, May 1998.

[7] D. C. Burger and T. M. Austin. The SimpleScalar Tool Set. Computer Architecture News, 1997. See also `http://www.simplescalar.com` for more information.

[8] F. J. Cazorla, E. Fernandez, A. Ramirez, and M. Valero. Optimizing long-latency-load-aware fetch policies for SMT processors.

International Journal of High Performance Computing and Networking (IJHPCN), 2(1):45–54, 2004.

[9] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernández, A. Ramirez, and M. Valero. Predictable performance in SMT processors: Synergy between the OS and SMTs. *IEEE Transactions on Computers*, 55(7):785–799, July 2006.

[10] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez. Dynamically controlled resource allocation in SMT processors. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 171–182, December 2004.

[11] F. J. Cazorla, A. Ramirez, M. Valero, P. M. W. Knijnenburg, R. Sakellariou, and E. Fernández. QoS for high-performance SMT processors in embedded systems. *IEEE Micro*, 24(4):24–31, July 2004.

[12] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip-multiprocessor architecture. In *Proceedings of the Eleventh International Symposium on High Performance Computer Architecture (HPCA)*, pages 340–351, February 2005.

[13] S. Choi and D. Yeung. Learning-based SMT processor resource distribution via hill-climbing. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*, pages 239–250, June 2006.

[14] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 76–87, June 2004.

[15] D. Christie. Developing the AMD-K5 architecture. *IEEE Micro*, 16(2):16–27, 1996.

[16] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 468–477, October 1996.

[17] E. Cota-Robles. *Priority Based Simultaneous Multi-Threading*, December 2003. United States Patent No. 6,658,447 B2.

[18] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. *ProfileMe*: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 1997.

[19] K. Diefendorff. Compaq chooses SMT for Alpha. *Microprocessor Report*, 13(16), 1999.

[20] P. K. Dubey, G. B. Adams III, and M. J. Flynn. Instruction window size trade-offs and characterization of program parallelism. *IEEE Transactions on Computers*, 43(4):431–442, April 1994.

[21] P. K. Dubey and M. J. Flynn. Optimal pipelining. *Journal of Parallel and Distributed Computing*, 8(1):10–19, January 1990.

[22] L. Eeckhout, R. H. Bell Jr., B. Stougie, K. De Bosschere, and L. K. John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 350–361, June 2004.

[23] L. Eeckhout and K. De Bosschere. Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 25–34, September 2001.

[24] A. El-Moursy and D. H. Albonesi. Front-end policies for improved issue efficiency in SMT processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 31–40, February 2003.

[25] P. G. Emma. Understanding some simple processor-performance limits. *IBM Journal of Research and Development*, 41(3):215–232, May 1997.

[26] P. G. Emma and E. S. Davidson. Characterization of branch and data dependencies in programs for evaluating pipeline performance. *IEEE Transactions on Computers*, 36(7):859–875, July 1987.

[27] P. G. Emma, J. W. Knightand J. H. Pomerence, T. R. Puzak, and R. N. Rechtschaffen. Simulation and analysis of a pipeline proces-

sor. In *Proceedings of the 21st Winter Simulation Conference*, pages 1047–1057, 1989.

[28] S. Eyerman and L. Eeckhout. A memory-level parallelism aware fetch policy for SMT processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 240–249, February 2007.

[29] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate CPI components. In *Proceedings of The Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–184, October 2006.

[30] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A top-down approach to architecting CPI component performance counters. *IEEE Micro*, 17(1):84–93, January 2007.

[31] S. Eyerman and L.Eeckhout. Studying compiler optimizations on superscalar processors through interval analysis. In *Proceedings of the 2008 International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, pages 114–129, January 2008.

[32] S. Eyerman, James E. Smith, and L. Eeckhout. Characterizing the branch misprediction penalty. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 48–58, March 2006.

[33] A. Fedorova, M. Seltzer, and M. D. Smith. A non-work-conserving operating system scheduler for SMT processors. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA), in conjunction with ISCA*, pages 134–145, June 2006.

[34] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 25–38, September 2007.

[35] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn. Interaction cost and shotgun profiling. *ACM Transactions on Architecture and Code Optimization*, 1(3):272–304, September 2004.

[36] R. Gabor, S. Weiss, and A. Mendelson. Fairness enforcement in switch on event multithreading. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(3):34, September 2007.

[37] D. Genbrugge and L. Eeckhout. Memory data flow modeling in statistical simulation for the efficient exploration of microprocessor design spaces. *IEEE Transactions on Computers*, 57(1):41–54, January 2008.

[38] A. Glew. MLP yes! ILP no! In *ASPLOS Wild and Crazy Idea Session*, October 1998.

[39] S. Gochman, A. Mendelson, A. Naveh, and E. Rotem. Introduction to Intel Core Duo processor architecture. *Intel Technology Journal*, 10(2):89–98, 2006.

[40] G. F. Grohoski. Machine organization of the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 36(1):37–58, 1990.

[41] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization (WWC)*, December 2001.

[42] G. Hamerly, E. Perelman, J. Lau, and B. Calder. SimPoint 3.0: Faster and more flexible program analysis. *Journal of Instruction-Level Parallelism*, 7, September 2005.

[43] A. Hartstein and T. R. Puzak. The optimal pipeline depth for a microprocessor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 7–13, May 2002.

[44] A. Hartstein and T. R. Puzak. Optimum power/performance pipeline depth. In *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO)*, pages 117–126, December 2003.

[45] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[46] M. S. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. Burger, S. W. Keckler, and P. Shivakumar. The optimal logic depth per pipeline stage

is 6 to 8 FO4 inverter delays. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 14–24, June 2002.

[47] Intel. *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, May 2004. 251110-003.

[48] E. Ipek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana. Efficiently exploring architectural design spaces via predictive modeling. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 195–206, October 2006.

[49] R. Iyer. CQoS: A framework for enabling QoS in shared caches of CMP platforms. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 257–266, June 2004.

[50] R. Iyer, L. Zhao, F. Guo amd R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 25–36, June 2007.

[51] R. Jain, C. J. Hughes, and S. V. Adve. Soft real-time scheduling on simultaneous multithreaded processors. In *Proceedings of the 23rd IEEE International Real-Time Systems Symposium*, pages 134–145, December 2002.

[52] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. Construction and use of linear regression models for processor performance analysis. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 99–108, February 2006.

[53] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. A predictive performance model for superscalar processors. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 161–170, December 2006.

[54] T. Karkhanis and J. E. Smith. A day in the life of a data cache miss. In *Proceedings of the 2nd Annual Workshop on Memory Performance Issues (WMPI) held in conjunction with ISCA*, May 2002.

[55] T. Karkhanis and J. E. Smith. Automated design of application specific superscalar processors: An analytical approach. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pages 402–411, June 2007.

[56] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 338–349, June 2004.

[57] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance characterization of a quad Pentium Pro SMP using OLTP workloads. In *Proceedings of the 25th International Symposium on Computer Architecture (ISCA)*, June 1998.

[58] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT))*, pages 111–122, September 2004.

[59] A. J. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1(2):10–13, June 2002.

[60] S. Kunkel and J. E. Smith. Optimal pipelining in supercomputers. In *Proceedings of the 13th Annual International Symposium on Computer Architecture (ISCA)*, pages 404–411, June 1986.

[61] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, 2007.

[62] B. Lee and D. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 185–194, October 2006.

[63] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 164–171, November 2001.

[64] Y. Luo, J. Rubio, L. K. John, P. Seshadri, and A. Mericas. Benchmarking internet servers on superscalar machines. *IEEE Computer*, 36(2):34–40, February 2003.

[65] A. Mericas. POWER5 performance measurement and characterization. Tutorial at the IEEE International Symposium on Workload Characterization, October 2005.

[66] A. Mericas. Performance monitoring on the POWER5 microprocessor. In L. K. John and L. Eeckhout, editors, *Performance Evaluation and Benchmarking*, pages 247–266. CRC Press, 2006.

[67] P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 2–10, October 1999.

[68] P. Michaud, A. Seznec, and S. Jourdan. An exploration of instruction fetch requirement in out-of-order superscalar processors. *International Journal on Parallel Programming*, 29(1), February 2001.

[69] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 2007.

[70] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 129–140, February 2003.

[71] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 208–222, December 2006.

[72] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 57–68, June 2007.

[73] D. B. Noonburg and J. P. Shen. Theoretical modeling of superscalar processor performance. In *Proceedings of the 27th Annual*

International Symposium on Microarchitecture (MICRO), pages 52–62, November 1994.

[74] D. B. Noonburg and J. P. Shen. A framework for statistical modeling of superscalar processor performance. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture (HPCA)*, pages 298–309, February 1997.

[75] S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 15–24, September 2001.

[76] M. Oskin, F. T. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor design. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, pages 71–82, June 2000.

[77] D.B. Papworth. Tuning the Pentium Pro microarchitecture. *IEEE Micro*, 16(2):8–15, 1996.

[78] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 244–256, September 2003.

[79] S. E. Raasch and S. K. Reinhardt. The impact of resource partitioning on SMT processors. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 15–26, September 2003.

[80] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1998.

[81] E. M. Riseman and C. C. Foster. The inhibition of potential parallelism by conditional jumps. *IEEE Transactions on Computers*, C-21(12):1405–1411, December 1972.

[82] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceed-*

ings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 45–57, October 2002.

[83] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4-5):505–522, 2005.

[84] J. E. Smith and G. S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12):1609–1624, December 1995.

[85] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for simultaneous multithreading processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Ooperating Systems (ASPLOS)*, pages 234–244, November 2000.

[86] A. Snavely, D. M. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 66–76, June 2002.

[87] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood. Analytic evaluation of shared-memory systems with ILP processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA)*, pages 380–391, June 1998.

[88] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 25–34, May 2002.

[89] B. Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82, July 2002.

[90] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, R. N. Strenski, and P. G. Emma. Optimizing pipelines for power and performance. In *Proceedings of the 35th Annual International Symposium on Microarchitecture (MICRO)*, pages 333–344, December 2002.

[91] W. Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, fifth edition, 2005.

[92] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of the 1993 ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 24–35, 1993.

[93] T. M. Taha and D. S. Wills. An instruction throughput model of superscalar processors. In *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP)*, pages 156–163, June 2003.

[94] T. M. Taha and D. S. Wills. An instruction throughput model of superscalar processors. *IEEE Transactions on Computers*, 57(3):389–403, march 2008.

[95] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *Proceedings of the 22nd Annual Computer Measurement Group Conference*, December 1996.

[96] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 318–327, December 2001.

[97] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA)*, pages 191–202, May 1996.

[98] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 392–403, June 1995.

[99] M. G. Valluri and R. Govindarajan. Evaluating register allocation and instruction scheduling techniques in out-of-order issue processors. In *Proceedings of the 8th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 78–83, October 1999.

[100] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of the fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 176–188, April 1991.

[101] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 84–95, June 2003.

[102] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and comparing prevailing simulation techniques. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 266–277, February 2005.

[103] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance analysis using the MIPS R10000 performance counters. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, January 1996.